МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Є. Ю. БЕРКУНСЬКИЙ, А. Ю. ПАВЛЕНКО

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ МОВАМИ KOTLIN, C/C++

Навчальний посібник

Рекомендовано Вченою радою НУК

ВИДАВНИЦТВО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КОРАБЛЕБУДУВАННЯ
ІМ. АДМІРАЛА МАКАРОВА

2022

УДК 004.43(076)

Б48

Автори:
Є. Ю. Беркунський, старш. викладач;
А. Ю. Павленко, старш. викладач

Рецензенти:
І. І. Коваленко, д-р техн. наук, професор кафедри інженерії програмного забезпечення Чорноморського національного університету імені П. Могили;
С. Б. Приходько, д-р техн. наук, проф., завідувач кафедри програмного забезпечення автоматизованих систем Національного університету кораблебудування імені адмірала Макарова;
Т. В. Тихонова, д-р пед. наук, доц., завідувач кафедри педагогіки, психології та менеджменту освіти Миколаївського обласного інституту післядипломної педагогічної освіти

Рекомендовано Вченою радою НУК ім. адмірала Макарова як навчальний посібник
(*протокол № 06 від 25.06.2021 р.*)

Беркунський Є. Ю.

Б48 Алгоритмізація та програмування мовами Kotlin, C/C++ : навчальний посібник / Є. Ю. Беркунський, А. Ю. Павленко. – Миколаїв : НУК, 2022. – 256 с.

ISBN 978-966-321-446-7

Наведені теоретичні відомості та довідкова інформація для вивчення алгоритмізації, основ програмування та виконання завдань лабораторних робіт з відповідних дисциплін; основні відомості про середовища розробки компанії JetBrains: IntelliJ IDEA та CLion, структуру програм мовами Kotlin, C/C++, оператори, службові слова, інші елементи мови програмування. Визначені завдання та контрольні питання для лабораторних робіт.

Призначено для студентів спеціальностей 122 «Комп'ютерні науки», 124 «Системний аналіз», 126 «Інформаційні системи та технології» усіх форм навчання. Може бути корисним для самостійного навчання програмуванню.

УДК 004.43(076)

ISBN 978-966-321-446-7

ВСТУП

Навчальний посібник призначений для допомоги студентам під час вивчення дисциплін «Алгоритмізація та програмування», «Основи програмування» та вивчення основ розробки програм мовами Kotlin та C/C++. Головною особливістю посібника є те, що для вивчення основ програмування обрано сучасну мову програмування Kotlin. Після вивчення основ програмування мовою Kotlin, у темах 1–6 розглядаються базові алгоритмічні структури мовами C/C++ (тема 7), та надалі всі теми з 8 до 13 розглядаються з використанням Kotlin та C/C++.

Мова Kotlin – сучасна мова програмування, яка створена у 2011 році компанією JetBrains та використовується для розробки різноманітних програм від мобільних застосувань до серверного програмного забезпечення [1]. У 2018 році компанія Google зробила заяву, що розглядає Kotlin як основну мову програмування для розробки на платформі Android.

Для вивчення дисципліни використовуються матеріали лекцій та лабораторних практикумів, що знаходяться на сторінці https://github.com/kafedra-iust/. Перед виконанням завдань до кожного лабораторного практикуму студенту необхідно вивчити відповідний матеріал лекції та звернути особливу увагу на теоретичні відомості, що передують опису завдань. Наведені приклади програм не слід розглядати

як єдиний можливий варіант розв'язання завдань. Посібник складається з 13 тем, які вивчаються в рамках дисциплін «Алгоритмізація та програмування» та «Основи програмування» першого року навчання зі спеціальностей 122 «Комп'ютерні науки», 124 «Системний аналіз», 126 «Інформаційні системи та технології» усіх форм навчання в Національному університеті кораблебудування імені адмірала Макарова. Кожна тема відповідає окремій лабораторній роботі та складається з таких частин: основні теоретичні відомості, лабораторний практикум, що містить мету роботи, завдання та контрольні питання. Під час проведення всіх лабораторних практикумів рекомендуються до використання такі засоби: персональна ЕОМ з процесором класу Intel 8 покоління або більш новим, операційна система Microsoft Windows 10 або Linux (наприклад, Linux Mint), середовища програмування від компанії JetBrains: IntelliJ IDEA та CLion. Дозволяється використання інших програмних засобів. Під час виконання робіт лабораторних практикумів студент повинен продемонструвати: творчий, індивідуальний підхід до розробки проєктів (програмного коду); грамотне використання існуючого програмного забезпечення; навички програмування мовами високого рівня Kotlin та C++. Студент повинен уміти перетворити алгоритм у програмний продукт, використовувати якісний аналіз програми, виконувати оцінку одержаних результатів. Велике значення має зручний інтерфейс користувача й пояснення до програми.

Варіант завдання до лабораторних практикумів визначається викладачем.

Типовий порядок виконання робіт та методичні рекомендації до їх виконання:

1. Уважно ознайомитися з методичними рекомендаціями до конкретного лабораторного практикуму (теоретичними відомостями, прикладами, формулюванням завдань),

використовуючи засоби середовища розробки, створити мінімальне консольне застосування.

2. Доповнити мінімальне застосування конкретним змістом відповідно до запропонованого завдання (див. приклади); відлагодити програму, усуваючи всі помилки, що виникли на етапі компіляції початкового тексту програми.

3. Виконання програми здійснити на даних, що наведені в завданні до лабораторної роботи, та на власних даних.

4. Знайти свою папку проєкту й ознайомитися з її вмістом.

5. Виконати підсумковий запуск застосування за допомогою виконуваного модуля.

6. Відповісти на контрольні питання.

7. Оформити звіт і здати викладачеві.

Тема 1

Частина 1. Знайомство з інтегрованим середовищем розробки програм IntelliJ IDEA

Розробка простого консольного застосування

Запустіть IntelliJ IDEA, для чого виберіть відповідне посилання в Головному меню ОС, або двічі клацніть мишкою по ярлику застосування, якщо він виведений на робочий стіл. Після цього на екрані з'явиться вітальне (Welcome) вікно інтегрованого середовища розробки IntelliJ IDEA (рис. 1.1). Ви можете налаштувати його, обравши світлу або темну тему в розділі Customize.

Для створення консольного застосування оберіть New Project. З'явиться діалогове вікно New Project (рис. 1.2).

У цьому вікні оберіть тип проєкту (New Project), укажіть ім'я проєкту (Name:), оберіть місце розміщення проєкту (Location:) та систему збирання (Build System – наприклад, IntelliJ), та JDK (наприклад, 17). Нижче увімкніть прапорець «Add sample code». Усі інші налаштування залишіть без змін та натисніть кнопку «Create». IntelliJ IDEA створить порожній проєкт та відкриє його в головному вікні (рис. 1.3).

Відкрийте файл main.kt, який знаходиться в каталозі src/main/kotlin. У вікні, що з'явиться, буде відображено текст простої програми. Його можна змінити на такий (дещо спрощений):

```
fun main() {
    println("Hello World!")
}
```

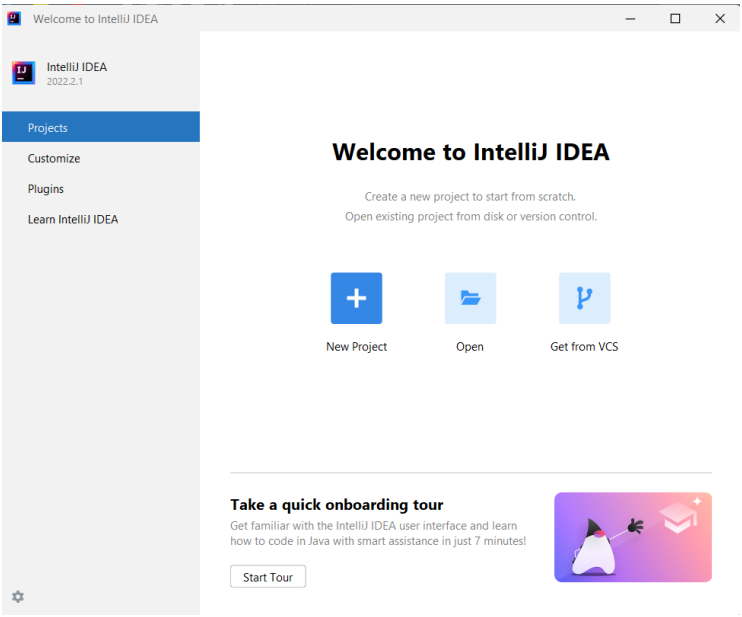


Рис. 1.1. Стартове (вітальне) вікно IDEA

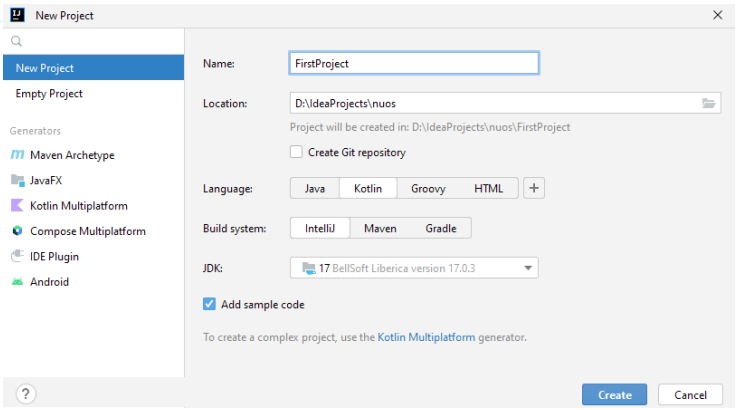


Рис. 1.2. Створення консольної програми мовою Kotlin

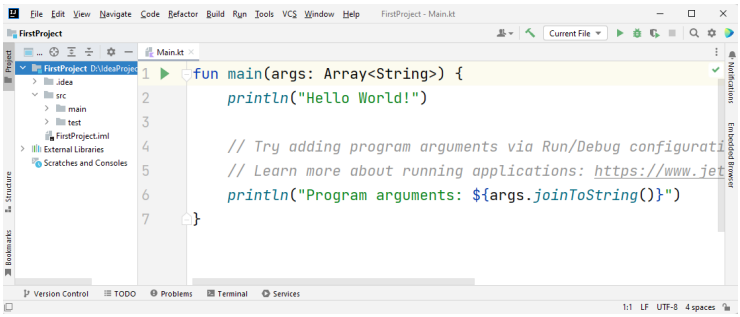


Рис. 1.3. Стартовий шаблон програми мовою Kotlin

Запустіть програму на виконання, клацнувши мишкою зелений трикутник, або натисніть Ctrl+Shift+F10. У вікні виведення ви побачите результат роботи програми – вітання `"Hello World!"`

Рис. 1.4. Результат роботи програми

Лабораторний практикум № 1

Частина 1

Мета: набути первинних практичних навичок роботи в середовищі програмування IntelliJ IDEA.

Порядок виконання роботи й методичні рекомендації до її виконання:

1. Створити на робочому диску папку для проєктів (ім'я папки повинно містити вичерпну інформацію про її користувача); запустити застосування IntelliJ IDEA.
2. Ознайомитися з елементами вікна IntelliJ IDEA.
3. Ознайомитися з командами кожного пункту головного меню.

4. Ознайомитися з елементами управління вікном IntelliJ IDEA.

5. Отримати у викладача навчально-демонстраційну програму або скористатися програмою, яка використовується в попередньому описанні середовища.

6. Виконати дії з введення, редагування, налагодження та збирання виконуваного модуля навчально-демонстраційної програми.

7. Підсумковий запуск виконуваного модуля виконати з командного рядка.

Контрольні питання

1. Для чого потрібні мови програмування?
2. Для чого потрібне середовище програмування (IDE)?
3. З яких компонентів складається IDE?

Тема 1

Частина 2. Розв'язання задач лінійного характеру. Алфавіт мови Kotlin, ідентифікатори та ключові слова

Для програмування завдань лінійного характеру, у яких операції виконуються в природному порядку, тобто в порядку їх запису в програмі, необхідно знати такі конструкції мови Kotlin: ідентифікатори, службові слова, описи даних, вирази, оператори, вбудовані функції. Вирази в мові Kotlin записуються за допомогою 26 рядкових та 26 прописних літер англійського алфавіту: `a b c d e f g h i j k l m n o p q r s t u v w x y z, A B C D E F G H I J K L M N O P Q R S T U V W X Y Z` (припустимі також літери кирилиці); десяти цифр: `0 1 2 3 4 5 6 7 8 9`; таких спеціальних символів: `+ - * / =, . _ : ; ? \ « ’~ | ! # $ & ( ) [ ] { } ^ @.`

До спеціальних символів відноситься також пропуск. Комбінації деяких символів, не розділених пропусками, інтерпретуються як один значущий символ: `++ — || && << >> >= <= == != += -= = /= .?: :: / */ //`

Ідентифікаторами називаються імена, що надають змінним, константам, типам даних і функціям, які використовуються в програмах. Після опису ідентифікатора, можна посилатися на об'єкт, що позначається ним. *Ідентифікатор* – це послідовність символів довільної довжини, яка містить літери, цифри й символи підкреслення, обов'язково починається з літери або символу підкреслення. У Kotlin ураховується регістр букв. Компілятор сприймає прописні й рядкові букви, як різні символи. Так, змінні `userName` та `UserName` розглядаються як два різні ідентифікатори.

Ключові слова є зарезервованими ідентифікаторами, кожному з яких відповідає певна дія. Змінити призначення ключового слова неможливо. Імена ідентифікаторів, що створюються в програмі, не повинні збігатися з ключовими словами мови Kotlin (іноді дозволяється використовувати як ідентифікатори `soft keywords` та `modifiers`, проте цього краще не робити) [1].

Hard Keywords

as	break	class	continue	do	else	false
for	fun	if	in	interface	is	null
object	package	return	super	this	throw	true
try	typealias	typeof	val	var	when	while

Soft Keywords

by	catch	constructor	delegate	dynamic	field
file	finally	get	import	init	param
property	receiver	set	setparam	where	

Modifiers

`actual`	`abstract`	`annotation`	`companion`	`const`	`crossinline`
`data`	`enum`	`expect`	`external`	`final`	`infix`
`inline`	`inner`	`internal`	`lateinit`	`noinline`	`open`
`operator`	`out`	`override`	`private`	`protected`	`public`
`reified`	`sealed`	`suspend`	`tailrec`	`vararg`	

Стандартні типи даних, модифікатори, кваліфікатори доступу й перетворення типів

Кожна програма обробляє певну інформацію. У Kotlin дані мають один з базових типів: `Char` (текстові дані), `Int` (цілі числа), `Float` (числа з плаваючою точкою одинарної точності), `Double` (числа з плаваючою точкою подвійної точності), `Unit` (порожні значення), `Boolean` (логічні значення) та інші. Текстом (тип даних Char) є один символ. Зазвичай кожен символ займає 16 біт або два байти. Цілі числа (тип даних `Int`) знаходяться в діапазоні від `–2147483648` до `2147483647`. У Kotlin підтримуються чотири типи цілих чисел. Разом із стандартним типом `Int` існують типи `Byte, Short, Long`. Числа з плаваючою точкою одинарної точності (тип даних `Float`) можуть бути представлені як у фіксованому форматі, так і в експоненціальному. Діапазон значень – від ±3.4E-38 до ±3.4E+38, розмірність – 32 біта, тобто 4 байти або 2 слова. Числа з плаваючою комою подвійної точності (тип даних `Double`) мають діапазон значень від ±1.7E-308 до ±1.7E+308 і розмірності 64 біт, тобто 8 байтів або 4 слова. Тип даних `Unit` зазвичай застосовується у функціях, що не повертають ніякого значення. Змінні логічного типу даних `Boolean` у Kotlin можуть містити тільки одну з двох констант: `true` або `false`. Іноді потрібно, щоб значення змінної залишалося постійним протягом усього часу існування змінної. Такі змінні

називаються *константними.* Наприклад, якщо в програмі обчислюється довжина кола або площа круга, часто доводиться оперувати числом π (3,1415926). Для оголошення константних змінних використовується ключове слово **val** у той час, як для інших змінних (що мутують) – **var**. Часто буває, коли в операції беруть участь змінні різних типів. Такі операції називаються *змішаними*. Деякі з них дозволені, а деякі – заборонені. Наприклад:

```
var a = 2
var res = 3.7
a = a * res      //Помилка!
```

У процесі виконання змішаних операцій компілятор намагається автоматично проводити перетворення типів даних. Цілочисельне значення змінної a зчитується з пам'яті, приводиться до типу з плаваючою точкою та помножується на початкове значення змінної res, отримуємо 7,4. Результат у вигляді значення з плаваючою точкою присвоюється змінній цілого типу a, отримуємо помилку через звужуюче перетворення. Автоматичні перетворення типів даних під час виконання змішаних операцій здійснюються відповідно до ієрархії перетворень. Суть полягає в тому, що з метою підвищення продуктивності, у змішаних операціях значення різних типів тимчасово приводяться до того типу даних, який має більший пріоритет в ієрархії. Нижче перераховані типи даних у порядку зниження пріоритету: `Double, Float, Long, Int, Short, Byte`.

Якщо значення перетвориться на тип, що має більшу розмірність, не буде мати місця втрата інформації, унаслідок чого не страждає точність обчислень, такі автоматичні перетворення дозволяються. Наприклад:

```
var a = 2
var res = 3.7
res = a * res      // операція дозволена
```

Іноді потрібно змінити тип змінної, не чекаючи автоматичного перетворення. Для цього призначена операція приведення типу. Якщо в програмі необхідно тимчасово змінити тип змінної, потрібно явно викликати операцію перетворення до відповідного типу даних. Наприклад:

```
r = v + (a / b).toFloat()
r = v + a / b.toFloat()
r = v + a.toFloat() / b.toFloat()
```

У всіх трьох випадках перед виконанням ділення відбувається явне приведення значення однієї або двох змінних до типу `Float`.

Операції

Kotlin включає в себе побітові операції, операції інкрементування і декрементування, умовну операцію, операції комбінованого присвоєння. Побітові операції працюють із змінними як із наборами бітів, а не як із числами. Ці операції використовуються в тих випадках, коли необхідно отримати доступ до окремих бітів даних (наприклад, під час виведення графічних зображень на екран). Побітові операції застосовуються тільки до цілочисельних значень. На відміну від логічних операцій, із їх використанням порівнюються не два числа цілком, а окремі їхні біти. Основні побітові операції: «І» (`and`), «АБО» (`or`) і «Виключне АБО» (`xor`). Сюди можна також зарахувати унарну операцію побітового інвертування (`inv`), яка інвертує значення бітів числа.

Операція `and` записує в біт результату одиницю тільки в тому випадку, якщо обидва порівнюваних біта дорівнюють 1. Ця операція часто використовується для маскування окремих бітів числа. Наприклад: `0xF1` **`and`** 0x35 = 0x31.

Операція **or** записує в біт результату одиницю в тому випадку, якщо хоча б один з порівнюваних бітів дорівнює 1. Ця операція часто застосовується для установлення окремих бітів числа. Наприклад: `0xF1` **`or`** 0x35 = 0xF5.

Операція **xor** записує в біт результату одиницю в тому випадку, якщо порівнювані біти відрізняються один від одного. Ця операція часто застосовується під час виведення зображень на екран, коли відбувається накладення декількох графічних шарів. Наприклад: `0xF1` **`xor`** 0x35 = 0xC4.

Таблиця 1.1. **Логічні операції в Kotlin**

A	B	a and b	a or b	a xor b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```
var a = 65          // молодший байт: 01000001
a = a shl 1         // молодший байт: 10000010
println(a)          // буде виведене 130
```

Зсув праворуч супроводжується аналогічними діями, тільки бітове представлення числа зрушується на вказану кількість бітів управо. Значення молодших бітів втрачаються, а старші біти, що звільнилися, заповнюються нулями, якщо операнд беззнаковий, і значенням знакового біта інакше. Таким чином, зсув беззнакового числа на одну позицію праворуч еквівалентне діленню числа на два:

```
var a = 10          // молодший байт: 00001010
a = a shr 1         // молодший байт: 00000101
println(a)          // буде виведене 5
```

Збільшення (зменшення) значення змінної на 1 дуже часто зустрічається в програмах, тому розробники мови Kotlin передбачили для цих цілей спеціальні операції інкрементування (`++`) і декрементування (`--`). Так, замість рядка `a+1`, можна ввести рядок `a` або `++a`.

За ситуації, коли операція є єдиною у виразі, не має значення місце її розташування: до імені змінної або після нього. Значення змінної в будь-якому випадку збільшиться

на одиницю. У процесі роботи зі складними виразами необхідно уважно стежити, коли саме відбувається модифікація змінної. Потрібно розрізняти префіксні й постфіксні операції, які ставляться відповідно до або після імені змінної. Наприклад, при постфіксному інкрементуванні і спочатку повертається значення змінної, після чого воно збільшується на одиницю. З іншого боку, операція префіксного інкрементування `++i` вказує, що спочатку слід збільшити значення змінної, а потім повернути його як результат. Наприклад: нехай i=3, тоді

```
k = ++i  // набувають значення i=4, k=4
k = i++   //спочатку k=4, потім і збільшиться на
одиницю (i=5)
k = --i   //спочатку зменшиться на одиницю i=4, k=4
k = i--   //k=4,i=3
```

У Kotlin представлені всі стандартні арифметичні операції: складання (+), віднімання (–), множення (*), ділення (/) і ділення по модулю (%). Перші чотири операції не вимагають роз'яснень. Суть операції ділення по модулю:

```
var a = 3
var b = 8
var d = b % a // результат: 2
```

При діленні по модулю повертається залишок від операції ділення націло.

Також у мові Kotlin присутні комбіновані операції, що становляють собою скорочені еквіваленти запису звичайних операцій:

Таблиця 1.2. **Комбіновані операції в Kotlin**

Початковий оператор	Еквівалент	Коментар
`v = v + 3`	`v += 3`	До змінної додається 3
`v = v - 10`	`v -= 10`	Від змінної віднімається 10
`v = v * 3.14`	`v *= 3.14`	Змінна помножується на 3.14

Продовж. табл. 1.2

Початковий оператор	Еквівалент	Коментар
`v = v / 2.5`	`v /= 2.5`	Змінна ділиться на 2.5
`v = v % 2`	`v %= 2`	Отримання залишку при діленні v на 2
`v = v + 1`	`v++`	Операція інкремента
`v = v - 1`	`v--`	Операція декремента

Операції порівняння призначені для перевірки рівності або нерівності порівнюваних операндів. Усі вони повертають true в разі встановлення істинності виразу і false інакше. Нижче перераховані оператори порівняння, використовувані в мові Kotlin.

Таблиця 1.3. **Операції порівняння в Kotlin**

Операція	Виконувана перевірка
==	Дорівнює
!=	Не дорівнює
>	Більше
<	Менше
<=	Менше або дорівнює (не більше)
>=	Більше або дорівнює (не менше)

Логічні операції І (&&), АБО (||) і НЕ (!) повертають значення true або false залежно від логічного відношення між їх операндами. Так, операція && повертає true, коли істинні (не дорівнюють нулю) обидва його аргументи. Оператор || повертає false тільки в тому випадку, якщо обидва його аргументи не є істинними (дорівнюють нулю). Оператор ! інвертує значення свого операнду з false на true і навпаки. Послідовність виконання різних операцій визначається компілятором. Якщо не враховувати порядок розбору виразу компілятором, можуть бути одержані непра-

вильні результати. У таблиці перераховані всі операції мови Kotlin у порядку зниження їх пріоритету і вказаний напрям обчислення операндів (асоціативність): зліва направо або справа наліво.

Таблиця 1.4. **Знаки операцій в Kotlin**

Операція	Опис	Асоціативність
`++`	Постфіксний (префіксний) інкремент	Зліва направо
`—`	Постфіксний (префіксний) декремент	
`()`	Виклик функції	
`[]`	Доступ до елемента масиву	
`.`	Прямий доступ до члена класу	
`!`	Логічне НЕ	
`inv`	Побітове НЕ	
`-`	Унарний мінус	
`+`	Унарний плюс	
`*`	Множення	Зліва направо
`/`	Ділення	
`%`	Ділення по модулю	
`+`	Складання	Зліва направо
`-`	Віднімання	
`shl`	Зсув ліворуч	Зліва направо
`shr`	Зсув праворуч	
`<`	Менше	Зліва направо
`>`	Більше	
`<=`	Менше або дорівнює	
`>=`	Більше або дорівнює	
`==`	Дорівнює	Зліва направо
`!=`	Не дорівнює	
`and`	Побітове І	Зліва направо
`xor`	Побітове виключаюче АБО	Зліва направо
`or`	Побітове АБО	Зліва направо
`&&`	Логічне І	Зліва направо
`||`	Логічне АБО	Зліва направо

Продовж. табл. 1.4

Операція	Опис	Асоціативність
=	Просте присвоювання	Справа наліво
*=	Присвоювання з множенням	
/=	Присвоювання з діленням	
%=	Присвоювання з діленням по модулю	
+=	Присвоювання з додаванням	
−=	Присвоювання з відніманням	

Стандартні математичні функції мови Kotlin

У мові Kotlin усі стандартні функції знаходяться в бібліотеках, які можна підключити за допомогою імпорту з пакетів Kotlin та/або Java. Обчислення в програмах на Kotlin неможливі без використання математичних функцій, які описані у файлі пакету kotlin.math (або у класі java.lang. Math). Оскільки в мові Kotlin є багато стандартних математичних функцій, тут не будемо їх наводити, а наведемо посилання на офіційний сайт: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.math/index.html.

Розглянемо приклад: знаходження значення похідної функції в точці.

Постановка завдання: задана функція $y = sin(x)$. Знайти її похідну в точці $x = \pi / 2$. Для знаходження похідної в точці використовується відомий вираз:

$$y'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

```
import kotlin.math.sin

fun main() {
    val dx = 1.0e-11
    val x = 3.1415926
    val f1 = sin(x+dx)
```

```
    val f2 = sin(x)
    val pf = (f1-f2)/dx
    println("dsin(x)/dx = $pf x = $x")
}
```

Операції консольного введення в Kotlin

Майже кожна програма, перед тим як обробляти дані, повинна отримати їх від користувача. У мові Kotlin для цього існує багато засобів. Розглянемо деякі з них.

Стандартна бібліотечна функція `readLine()` призначена для читання рядка з клавіатури. Таким чином можна виконувати введення даних за принципом – по одному в рядку:

```
val x=readLine()!!.toDouble() //введення дійсного числа
val n = readLine()!!.toInt()   // введення цілого числа
val s = readline()!!              // введення рядка
```

Якщо ж потрібно прочитати деяку кількість однотипних даних з одного рядка, які відокремлені пробілами, то це можна зробити, наприклад, так:

```
val (a,b,c) = readLine()!!.split(" ")
    .map(String::toInt)
val (x,y) = readLine()!!.split(" ")
    .map(String::toDouble)
```

Також, за потреби, можна скористатися можливостями, що надає клас Scanner стандартної бібліотеки Java:

```
val scanner = Scanner(System.in)
...
val a = scanner.nextInt()
val x = scanner.nextDouble()
```

Операції форматного виведення в Kotlin

У попередніх прикладах для виведення даних використовувалися так звані рядкові шаблони – рядки, у які були вставлені позначки – посилання на значення змінних. Наприклад:

```
println("dsin(x)/dx = $pf x = $x")
```

де `$pf` та `$x` – посилання на значення змінних `pf` та `x`. У відповідних місцях рядка, що виводиметься, будуть вставлені значення цих змінних. Але такий підхід є не дуже зручним через те, що таким чином не можна контролювати точність та кількість знаків, що використовується під час виведення.

Для цієї мети Kotlin використовує так звані «форматні шаблони».

Форматні шаблони для виведення звичайних, символьних та числових типів мають наступний синтаксис:

`%[індекс_аргумента$][опції][ширина][.точність]перетворення`

Необов'язковий параметр індекс_аргумента є цілим числом, що вказує позицію у списку аргументів. Посилання на перший аргумент буде записане як "`1$`", на другий – "`2$`" і т. д.

Необов'язковий параметр опції – це набір символів, що змінюють формат виведення. Набір припустимих опцій залежить від типу перетворення.

Необов'язковий параметр ширина – це невід'ємне ціле число, що показує мінімальну кількість символів, що їх треба вивести.

Необов'язковий параметр точність – це невід'ємне ціле число, що зазвичай використовується для обмеження кількості символів, що будуть виведені. Його дія залежить від параметра перетворення.

Обов'язковий параметр перетворення – це один символ, що вказує як аргумент буде відформатований. Набір припустимих перетворень для вказаного аргументу залежить від типу даних аргументу.

Оскільки синтаксис форматних шаблонів був запозичений з мови Java, інформацію про всі типи перетворень можна знайти на сайті Oracle, у розділі присвяченому мові програмування Java [1].

Тут наведемо лише декілька найпоширеніших прикладів:

```
println("Hello, World!")                    // Hello, World!
println("Sum %d + %d = %d".format(a,b,a+b)) // Sum 15 + 2 = 17
println("Pi =%5.2f".format(Math.PI))        // Pi = 3,14
```

Лабораторний практикум № 1

Частина 2

Мета: набути практичних навичок з підготовки, налагодження та виконання лінійних програм.

Завдання

Завдання 1.1

Записати мовою Kotlin математичні вирази, надані в табл. 1.5.

Таблиця 1.5.

Варіанти	Вираз 1	Вираз 2
1–3	$\frac{ln\lvert 2z\rvert + arctg 2z^2}{3(z+1)^2 + 2{,}1\cdot 10^6}$	$ln\lvert x+z\rvert > 0$ та $0 < b < 1$
4–6	$\frac{ln\lvert 5z\rvert + arctg^2 z}{3(z+1)^2 + 2{,}1\cdot 10^{-6}}$	$ln\lvert x+z\rvert >$ або $0 < b < 1$
7–9	$\frac{10^{-7} ln\lvert 2z\rvert + sin 2z^3}{3(z+3)^2 + 2{,}1\cdot 10^7}$	$\lvert x+z\rvert > 1$ та $1 < b < 2$
10–12	$\frac{10^{-7} ln\lvert 2z\rvert + b^{0,4}}{ln(z+1)^2 + 4{,}2\cdot 10^4}$	$\lvert x\rvert > 2$ або $0 < b < 3$
13–15	$\frac{10^{-5} e^{-5f} + sin^2\lvert z^3\rvert}{5(z+1)^5 + 10^6}$	$x + z < 0$ або $0 < f < 0{,}2$

Продовж. табл. 1.5

Варіанти	Вираз 1	Вираз 2
16–18	$\frac{10^{-4}e^{-2f}+ln\lvert z^3\rvert}{2(z+2)^{1,5}}$	$cos\lvert x+z\rvert>0$ або $0<b<3$
19–21	$\frac{ln\lvert 3z\rvert+arctg2z^2}{3(z+1)^2+2,1\cdot10^6}$	$\lvert x+z\rvert>0$ та $0<b<7$
22–24	$\frac{10^{-7}sin\lvert 3z\rvert+b^{1,2}}{ln(z+1)^2+1,2\cdot10^6}$	$ln\lvert x+z\rvert>0$ або $0<b<1$
25–27	$\frac{10^{-6}ln\lvert z^3\rvert+ln^2z^3}{6(z+1)^6+10^6}$	$cos\lvert x+z\rvert>0$ та $0<b<3$
28–30	$\frac{10^{-7}ln\lvert 3z^3\rvert+sin2z^2}{(z+1)^{0,5}+10^6}$	$\lvert x+z\rvert>0$ або $0<b<1$
31–33	$\frac{ln\lvert 4z\rvert+arctg^3 2z}{4(z+1)^{0,2}+1,7\cdot10^3}$	$\lvert x+z\rvert>0$ та $0<b<7$
34–36	$\frac{10^{-5}e^{-3f}+ln\lvert z^{-3}\rvert}{5(z+2)^{2,5}}$	$x+z<0$ та $0<f<0,2$

Завдання 1.2

Представити математичний запис виразу, що записаний мовою Kotlin (табл. 1.6), і показати порядок дій.

Таблиця 1.6.

Варіанти	Завдання
1, 19	`x+2.0/3.0/x/a+sqrt(sin(x))/2*sqrt(x)+1.0e-6*x.pow(1.0/7.0)`
2, 20	`(x+7)/3*x+3*atan(x)/2/x+1.0e7-sqrt(1.0/3.0*x.pow(5))`
3, 21	`x+2/3.0*x/a+sqrt(cos(x))/2/sqrt(x)+1.0e-5*x.pow(7)`
4, 22	`(x+4)/3/x+exp(abs(atan(x)))/2*x+1.0e-6*x.pow(1.0/3)`
5, 23	`x+2/3.0/x/a+sqrt(sin(x))/2.0/ln(x)+1.0e5*(x/3).pow(2/7.0)`

Продовж. табл. 1.6

Варіанти	Завдання
6, 24	`1.4e-4*(2*x).pow(3)+sqrt(sin(x))/2+sqrt(cos(x))/2/x`
7, 25	`sqrt(cos(x))/2/x-5.0/7.0*x/a/1.0e-6*(x/2).pow(1/3.0)*abs(x)`
8, 26	`x+2.0/3/x*a+sqrt(sin(x))/2/ln(x)+1.0e-3*(x/7).pow(2.0/3)`
9, 27	`(x+7)/3*x+3*atan(x)/2/x+1.0e7-sqrt(4*x.pow(b))`
10, 28	`sqrt(cos(abs(x)))/2/x-5.0/7*x/a/1.0e-6*(x/2).pow(1.0/8.0)`
11, 29	`x+9/(3*x/a)+sqrt(cos(x))/2/sqrt(x)+1.0e-5*x.pow(9)`
12, 30	`x+4/3.0/x+exp(abs(atan(x)))/2*x+1.0e-4*x.pow(1.0/3.0)`
13, 31	`sqrt(abs(cos(x)))/2/(x-5.0/7)x/a/1.0e-6(x/2).pow(5/3.0)`
14, 32	`x+2*3/x*a+ln(abs(sin(x)))/(2*cos(x)+1.0e-3*(x/2).pow(1.0/7))`
15, 33	`x+4.0/3/(x+abs(atan(x)))/2*x+1.0e-5*x.pow(5.0/3.0)`
16, 34	`sqrt(cos(x))/2*x-4.0/3*x/a/1.0e8*(x/3).pow(2.0/3.0)*sin(x)`
17, 35	`ln(x+5)/2*x+4*atan(x)/5/x+1.0e5-sqrt(4*x.pow(b/(2.0/3.0)))`
18, 36	`x+5.0/(3*x/a)+ln(abs(cos(x)))/2/exp(x)+1.0e-5*x.pow(3)`

Завдання 1.3

Скласти програму обчислення наступних величин та виконати її. *Позначення*: *N – номер варіанта за списком групи.*

Таблиця 1.7.

Варіанти	Умова
1	Модуль вектора 5**a**+10**b**, якщо **a**={3; 2} і **b**={0; -1}
2–6	Сума всіх парних чисел від 2 до 50***N**
7–11	Сума всіх двозначних цілих чисел, які кратні **N**
12	Кут між векторами **a**={1; 2} і **b**={1; -0,5}
13	Площа чотирикутника з вершинами A(0; 0), B(-1; 3), C(2; 4), D(3; 1)
14	Сума одинадцяти перших членів арифметичної прогресії, якщо $a_3+a_9=8$
15	Периметр трикутника з вершинами A(1; 1), B(4; 1), C(4; 5)
16	Модуль вектора -2**a** + 4**b**, якщо **a**={3; 2}, **b**={0; -1}
17	Кути трикутника з вершинами A(0; 1,7), B(2; 1,7), C(1,5; 0,85)

Продовж. табл. 1.7

Варіанти	Умова
18	Шостий член геометричної прогресії 5, -10, …
19	Кут між векторами a={2; -4; 4} та b={-3; 2; 6}
20	Модуль вектора **a**-**b**, якщо |**a**|=3, |**b**|=5 та вектори утворюють кут у 120º
21	Сума всіх двозначних цілих чисел
22	Модуль вектора **a**+**b**, якщо |**a**|=11, |**b**|=23, |**a**-**b**|=30
23	Сума всіх непарних двозначних чисел
24–28	Сума всіх тризначних цілих чисел, які в разі ділення на 5 дають остачу 28-**N**
29–32	Сума всіх непарних чисел від 3 до 5***N**
33–36	Сума всіх парних чисел від 10 до 7***N**

Контрольні питання

1. У чому різниця між іменами та ключовими словами?
2. Які засоби введення-виведення використовуються в Kotlin?
3. Які стандартні типи даних є в Kotlin?

Тема 2. ФУНКЦІЇ

Функцією в програмуванні називається закінчена ділянка програми, що вирішує певне завдання – зазвичай це частина великого завдання, яке вирішує програма загалом.

У простих випадках написання програми зводиться до написання однієї функції.

Як і у функції в математичному сенсі, у функції в програмуванні є входи (параметри), вихід (результат) і визначення, яке вказує, як розраховується значення виходу за заданим значенням входів.

Визначення простих функцій у мові Kotlin мало відрізняється від визначення математичних функцій. Розглянемо для прикладу математичну функцію $sqr(x) = x^2$. На Kotlin вона буде записана так:

```
fun sqr(x: Int) = x * x
// або
fun sqr(x: Double) = x * x
```

У цьому визначенні **fun** – це *ключове слово*, з якого починається визначення будь-якої функції в Kotlin; **fun** є скороченням від function–функція. sqr – це *ім'я* функції, x – *параметр* функції, = x * x – *тіло* функції, яке визначає, як треба обчислити її *результат*. Ім'я функції разом з її параметрами та ключовим словом **fun** називається її *заголовком*.

Оскільки *операція* обчислення квадрата числа в Kotlin відсутня, результат обчислюється як добуток x*x.

Математичні функції

У бібліотеці мови Kotlin визначена велика кількість математичних функцій, які призначені для виконання більш складних операцій. Для прикладу їхнього використання розглянемо задачу розв'язання квадратного рівняння: $ax^2 + bx + c = 0$.

Нагадаємо, що корені квадратного рівняння обчислюються за формулою

$$x_{1,2} = \frac{-b \pm \sqrt{d}}{2a},$$

де d – дискримінант квадратного рівняння – обчислюється як $d = b^2 - 4ac$.

Ми розв'яжемо цю задачу у спрощеному вигляді – знайти будь-який з двох можливих коренів, скажімо, той, у якому в чисельнику використовується знак плюс.

Для початку напишемо функцію для розрахунку дискримінанта (вона ще знадобиться нам у майбутньому). Для розрахунку b^2 використаємо вже написану вище функцію sqr(x: Double)

```
fun discriminant(a: Double, b: Double, c: Double) =
sqr(b) - 4 * a * c
```

У цьому фрагменті b є *аргументом* функції sqr. Запис виду sqr(b) називається викликом функції sqr. Підкреслимо відмінність *параметра* та *аргументу*: параметр визначається всередині функції та має визначене ім'я, у цьому випадку x, а аргумент передається у функцію ззовні та може бути як іменем змінної, так і більш складним виразом.

Тепер напишемо функцію для пошуку кореня квадратного рівняння. Для обчислення квадратного кореня застосуємо існуючу математичну функцію sqrt(x: Double) з математичної бібліотеки мови Kotlin

```
fun quadraticEquationRoot(a: Double, b: Double, c: Double) =
        (-b + sqrt(discriminant(a, b, c))) / (2 * a)
```

Тут ми зі свого боку використовуємо вже написану функцію discriminant для пошуку дискримінанта, і вираз discriminant(a, b, c), тобто дискримінант рівняння, є аргументом функції sqrt. Це саме той випадок, коли аргумент є складним виразом.

Змінні у функціях

Вище ми розглянули приклади з функціями sqr та discriminant, обчислення результату в яких займало один рядок коду. Проте, у програмуванні це – скоріш рідкий випадок; частіше розрахунок результату функції передбачає реалізацію деякої послідовності обчислень – алгоритму. Для збереження результатів проміжних обчислень програмісти використовують змінні.

Розглянемо, наприклад, задачу обчислення добутку двох коренів квадратного рівняння. Нагадаємо, що корені квадратного рівняння обчислюються як

$$\frac{-b+\sqrt{d}}{2a} \quad \text{та} \quad \frac{-b-\sqrt{d}}{2a}$$

відповідно, де d – дискримінант квадратного рівняння. При обчисленні добутку зручно спочатку зберегти обчислений корінь з дискримінанта у змінній **sd**, через те, що він використовується при обчисленні обох коренів. Після того треба обчислити обидва корені **x1** та **x2** і вже потім розрахувати їхній добуток. Мовою Kotlin це записується таким чином:

```
fun quadraticRootProduct(a: Double, b: Double, c:
Double): Double{
    /* тип обов'язковий */
    // Тіло функції у вигляді блока
    val sd = sqrt(discriminant(a, b, c))
    val x1 = (-b + sd) / (2 * a)
    val x2 = (-b - sd) / (2 * a)
    return x1 * x2 // Результат
}
```

У цьому прикладі тіло функції записано у вигляді блоку у фігурних дужках, у протилежність тілу у вигляді виразу, як у функціях sqr та discriminant вище. Знак рівності при цьому прибирається та обов'язково вказується тип результату функції. У прикладі присутні три проміжні змінні – d, x1, x2. Визначення проміжної змінної в Kotlin починається з ключового слова val (скорочення від value – значення), за яким іде ім'я змінної, та після знаку рівності, її значення. За бажанням можна також указати тип змінної, наприклад:

```
// ...
val sd: Double = sqrt(discriminant(a, b, c))
```

Якщо тип змінної не вказаний, він визначається автоматично, наприклад, у цьому випадку він співпадає з типом результату функції sqrt.

Блок складається з так званих операторів (у прикладі їх чотири), що виконуються по порядку згори донизу. Перед використанням будь-якої змінної, її слід визначити. Наприклад, такий запис призвів би до помилки:

```
fun quadraticRootProduct(a: Double, b: Double, c:
Double): Double {
     val x1 = (-b + sd) / (2 * a) // Unresolved
reference: sd
     val x2 = (-b - sd) / (2 * a) // Unresolved
reference: sd
    val sd = sqrt(discriminant(a, b, c))
    return x1 * x2 // Результат
}
```

Останній оператор функції, що починається з ключового слова **return**, визначає значення її результату; **return** перекладається з англійської як повернути (результат). Функція quadraticRootProduct насамперед обчислить значення змінної sd, використовуючи інші функції discriminant та sqrt. Потім відбудеться обчислення змінних x1 та x2 і лише в кінці – обчислення результату в операторі **return**.

Для порівняння, наведемо запис тієї ж функції, що не використовує змінні:

```
fun quadraticRootProduct(a: Double, b: Double, c:
Double) =
       ((-b + sqrt(discriminant(a, b, c))) / (2 * a)) *
       ((-b - sqrt(discriminant(a, b, c))) / (2 * a))
```

Хоча і записана в один рядок, така функція є набагато менш зрозумілою, під час її написання легко заплутатися при розстановці дужок. Крім того, у ній відбувається двократне обчислення кореня з дискримінанта, чого варто уникати.

Функція println та рядкові шаблони

Розглянемо приклад – функцію, що розв'язує квадратне рівняння та демонструє розв'язки користувачеві.

```
fun solveQuadraticEquation(a: Double, b: Double, c:
Double) {
    val sd = sqrt(discriminant(a, b, c))
    val x1 = (-b + sd) / (2 * a)
    val x2 = (-b - sd) / (2 * a)
    // Виведення на екран значень x1 та x2
    println(x1)
    println(x2)
   // Виведення на екран рядку вигляду x1 = 3.0 x2 = 2.0
    println("x1 = $x1 x2 = $x2")
    // Виведення на екран добутку коренів
    println("x1 * x2 = ${x1 * x2}")
}
```

Ми підійшли до такої важливої частини програмування, як взаємодія з користувачем та взагалі із зовнішнім для програми світом. Зверніть увагу – тепер функції, які ми використовуємо, починають відрізнятися від чисто математичних, оскільки в них з'являються побічні ефекти (side effects). Функція в програмуванні в загальному випадку не зводиться лише до залежності між параметрами та результатом.

Функція `println(p)` визначена у стандартній бібліотеці мови Kotlin, тому її застосування не потребує підключення будь-яких додаткових пакетів. Її параметр `p` може мати будь-який тип – так, виклик `println(x1)` виведе на окремий рядок консолі значення змінної `x1`.

Проте найчастіше p є рядком, наприклад, `"x1 = $x1 x2 = $x2"`. У цьому рядку присутні рядкові шаблони `$x1` та `$x2`, що складаються із символу `$` та імені змінної (параметра). Замість них програма автоматично підставить значення відповідних змінних.

Рядковий шаблон дозволяє також підставити значення складного виразу як, наприклад, тут: `"x1 * x2 = ${x1 * x2}"`. У цьому випадку вираз записується у фігурних дужках, щоб програма мала можливість відслідкувати його початок та кінець.

Зверніть увагу, що тип результату функції `solveQuadraticEquation` не вказаний. Це означає, що функція не має результату (у математичному сенсі). Такі функції зустрічаються доволі часто, один з прикладів – сама функція println, та їхній реальний результат зводиться до їхніх побічних ефектів – наприклад, виведення на консоль.

Залишилось визначити – що саме є *консоль*? У звичній нам операційній системі Windows *консоль* – це вікно або його частина, яку програма використовує для виведення текстової інформації. В IntelliJ IDEA це вікно можна відкрити послідовністю команд `View → Tool windows → Run`. Під час запуску програми з операційної системи, вона сама відкриє так зване «вікно термінала», яке буде використовуватися програмою для виведення текстової інформації.

Тестові функції

Тестові функції – особливий вид функцій, які призначені для перевірки правильності роботи інших функцій. Оскільки людині властиво помилятися, програмісти винайшли

чимало способів, як можна проконтролювати правильність програми, як своєї власної, так і написаної іншими людьми. Тестові функції є одним з таких способів. Написання тестових функцій вимагає підключення до програми однієї з бібліотек автоматичного тестування, наприклад, бібліотеки JUnit.

JUnit – бібліотека для тестування програмного забезпечення для мови Java. Створена Кентом Беком і Еріком Гаммою, JUnit є представником родини фреймворків xUnit для різних мов програмування, яка бере початок у SUnit Кента Бека для Smalltalk. JUnit породив екосистему розширень – JMock, EasyMock, DbUnit, HttpUnit, Selenium тощо.

Більшість класів цієї бібліотеки знаходяться в пакеті org.junit для версії JUnit 4.x або в пакеті org.junit.jupiter.api для версії JUnit 5.x.

Розглянемо приклад:

```
// Дозвіл на використання короткого імені анотації
// org.junit.jupiter.api.Test

import org.junit.jupiter.api.Test

// Дозвіл на використання короткого імені для функції
// org.junit.jupiter.api.Assertions.assertEquals

import org.junit.jupiter.api.Assertions.assertEquals

// Клас Tests, наявність класу обов'язкова для бібліотеки JUnit
class Tests {

    // ...

    // Тестова функція
    @Test
```

```
    fun testSqr() {
        assertEquals(0,sqr(0)) // Перевірити, що квад-
рат нуля = 0
        assertEquals(4,sqr(2)) // Перевірити, що квад-
рат двух = 4
       assertEquals(9,sqr(-3)) // Перевірити, що квад-
рат -3 = 9
    }
}
```

`@Test` – це так звана анотація, тобто позначка, яка використовується для надання функції `testSqr` додаткового сенсу. У цьому випадку анотація робить функцію `testSqr` тестовою. Функція `assertEquals` призначена для порівняння результату виклику деякої іншої функції, наприклад, `sqr`, з тим, що очікується. У наведеному прикладі вона викликається тричі.

Тестових функцій у проєкті може бути багато, будь-яка з них запускається так само, як і головна функція – натисканням зеленого трикутника зліва від заголовка функції. Тестові функції виконуються за тими ж принципами, що й будь-які інші, але виклики `assertEquals` відбуваються особливим чином:

• Якщо перевірка показала збіг результату з очікуваним, функція не робить нічого.

• В іншому випадку виконання тестової функції завершується і в IDEA з'явиться повідомлення, виділене червоним кольором, про невдале завершення тестової функції.

Якщо тестова функція завершила роботу й результати всіх перевірок співпали з очікуваними, тестова функція вважається завершеною успішно.

Нарешті, що ж таке `class Tests`? За правилами бібліотеки JUnit, усі тестові функції повинні бути присутніми

всередині деякого класу. Для чого взагалі потрібні класи, розглядатиметься в наступних роботах.

У цьому прикладі з цією метою був створений клас з ім'ям `Tests` (ім'я може бути довільним), тестова функція була записана в ньому. Зелений трикутник навпроти імені класу дозволяє одночасно запустити всі тестові функції в даному класі.

Будь-яка написана програма або функція завжди вимагає перевірки. Ця вимога тим важливіше, чим складніше програма або функція. Тестові функції дозволяють довести правильність роботи функції, яка перевіряється, принаймні для деяких значень її аргументів.

Поряд з тестовими функціями, може бути використано й ручне тестування. Ручне тестування передбачає виведення результатів функції на консоль і ручну перевірку їх з очікуваними. Для ручного тестування може бути використана головна функція, наприклад:

```
fun main() {
    println("sqr(0) = ${sqr(0)}")
    println("sqr(4) = ${sqr(4)}")
}
```

Якщо програма виконалася без помилок, ми повинні побачити на консолі рядки

```
sqr(0) = 0
sqr(4) = 16
```

Ручне тестування є набагато більш трудомістким і вимагає від програміста або тестувальника набагато більшої уваги. Тому в сучасному програмуванні рекомендується починати перевірку функцій зі створення тестових функцій, які запускаються кожен раз під час зміни програми й дозволяють помітити чи з'явилися помилки. Ручне тестування виконується значно рідше, зазвичай перед випуском нової версії програми.

Лабораторний практикум № 2

Мета: набути практичних навичок з підготовки, налагодження та виконання програм з використанням функцій.

Завдання

Завдання 2.1

Завантажте проєкт, що знаходиться на ресурсі GitHub за посиланням https://github.com/kafedra-iust/lab2kotlin.

Відкрийте файл `Main.kt`. Знайдіть у ньому описи заголовків функцій для свого варіанта. Наприклад, для 1 варіанта – це будуть функції:

```
fun var1calcR(a: Double, b: Double, x: Double) : Double = TODO()
fun var1calcS(a: Double, b: Double, x: Double) : Double = TODO()
```

Замість `TODO()` опишіть реалізацію цих функцій. Перейдіть до класу тестових функцій MainKtTest та виконайте тестування ваших функцій (табл. 2.1) за допомогою відповідних функцій цього класу.

Якщо функції тестування показали, що у Вас є помилки – виправіть їх (свої помилки, не функції!) та повторіть тестування.

Таблиця 2.1.

Варіант	Розрахункові формули	Значення вхідних даних
1	$r = x^2(x+1)/b - sin^2(x+a); s = \sqrt{\frac{xb}{a}} + cos^2(x+b)^3$	`a=0.7 b=0.05 x=0.5`
2	$f = \sqrt[3]{m \cdot tgt + \lvert csint \rvert}; z = mcos(btsint) + c$	`m=2;c=-1 t=1.2 b=0.7`
3	$y = btg^2 x - \frac{a}{sin^2(x/a)}; d = ae^{-\sqrt{a}} cos(bx/a)$	`a=3.2 b=17.5 x=-4.8`
4	$s = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}; f = x(sinx^3 + cos^2 y)$	`x=0.335 y=0.025`

Продовж. табл. 2.1

Варіант	Розрахункові формули	Значення вхідних даних
5	$s = x^3 tg^2\left(x+b\right)^2 + \frac{a}{\sqrt{x+b}}; Q = \frac{bx^2 - a}{e^{ax} - 1}$	`a=16.5 b=3.4 x=0.61`
6	$y = e^{-bt} sin\left(at+b\right) - \sqrt{\left\|bt+a\right\|}; s = bsin\left(at^2 cos 2t\right) - 1$	`a=-0.5 b=1.7 t=0.44`
7	$y = sin^3\left(x^2 + a\right)^2 - \sqrt{\frac{x}{b}}; z = \frac{x^2}{a} + cos\left(x+b\right)^3$	`a=1.1 b=0.004 x=0.2`
8	$a = \frac{2cos\left(x - \pi/6\right)}{1/2 + sin^2 y}; b = 1 + \frac{z^2}{3 + z^2/5}$	`x=1.426 y=-1.220 z=3.5`
9	$w = \sqrt{x^2 + b} - b^2 sin^3\left(x+a\right)/x; y = cos^2 x^3 - \frac{x}{\sqrt{a^2 + b^2}}$	`a=1.5 b=15.5 x=-2.8`
10	$c = \left\|x^{y/x} - \sqrt[3]{y/x}\right\|; f = \left(y-x\right)\frac{y - z/\left(y-x\right)}{1 + \left(y-x\right)^2}$	`x=1.825 y=18.225 z=-3.298`

Завдання 2.2

Для функцій, що були розроблені під час виконання завдання 2.1, створіть додаткові тести, виконайте тестування та впевніться в тому, що ці функції написані правильно.

Контрольні питання

1. Що таке функції в програмуванні?
2. Для чого потрібні функції в програмі?
3. З якої функції починається виконання програми мовою Kotlin?
4. Для чого потрібний оператор return?
5. Чи кожна функція повинна містити оператор return?
6. Що таке тестова функція?
7. Для чого потрібний тестовий фреймворк JUnit?
8. Яка особливість тестових функцій?
9. Наведіть приклади тестових функцій.
10. Що є результатом роботи тестової функції?

Тема 3. РОЗГАЛУЖЕННЯ

У попередніх роботах розглядали ситуації, коли обчислення результату функцій відбувалося без розгляду варіантів. Це можливий, але порівняно рідкісний випадок у програмуванні. Існує дуже багато завдань, де доводиться розглядати різні варіанти, і для цієї мети в мовах програмування існують конструкції розгалуження.

Максимум з двох

Як простий приклад розглянемо функцію, результат якої дорівнює найбільшому з її параметрів:

```
fun max(m: Int, n: Int) = if (m > n) m else n
```

Тут **if..else** – *оператор* або *конструкція* розгалуження, перекладається з англійської як **якщо..інакше**. Після ключового слова **if** в дужках іде *умова* розгалуження `m > n`. Якщо умова *істинна*, як результат використовується вираз одразу за умовою розгалуження, у цьому випадку `m`. Якщо ж умова *хибна*, використовується вираз після ключового слова **else**, у цьому випадку `n`. Конструкцію можна прочитати як «Якщо m > n, (то) m, інакше n». Функція `max`, аналогічна наведеній, існує у стандартній бібліотеці мови Kotlin.

Кількість коренів квадратного рівняння

Розглянемо більш складний приклад. Наступна функція обчислює кількість коренів квадратного рівняння:

$$ax^2 + bx + c = 0\,.$$

Нагадаємо, що квадратне рівняння має два корні, якщо його дискримінант більше 0, один корінь, якщо дискримінант дорівнює 0, і нуль коренів в іншому випадку. Для реалізації цього алгоритму слід спочатку розрахувати дискримінант, а потім застосувати конструкцію if..else. Функція мовою Kotlin може бути записана так:

```
fun quadraticRootNumber(a: Double, b: Double, c: Double): Int {
    // Застосовуємо готову функцію з попередньої роботи
    val d = discriminant(a, b, c)
    // Для порівняння на рівність застосовуємо ==
    return if (d > 0.0) 2 else if (d == 0.0) 1 else 0
}
```

Тут ми застосували запис функції у вигляді блоку та використали проміжну змінну d. Останній рядок функції читається як «повернути: якщо d > 0.0, (то) 2, інакше, якщо d = 0.0, (то) 1, інакше 0». Зверніть увагу на можливість так званого «каскадного» запису конструкції **if..else** у вигляді **if..else if..else if..else if..else** (з необмеженою кількістю проміжних елементів).

Операції порівняння

В обох прикладах в умовах ми використовували *операції порівняння*. Таких операцій є вісім:

- `>` Строго більше.
- `>=` Більше або дорівнює, аналог математичного ≥.
- `<` Строго менше.
- `<=` Менше або дорівнює, аналог математичного ≤.
- `X in a..b` – X належить інтервалу від a до b, аналог математичного $a \le x \le b$.
- `X ! in a..b` – X НЕ належить інтервалу від a до b.
- `!=` Не дорівнює, аналог математичного ≠.
- `==` Дорівнює (використовується два знаки рівності, щоб не плутати дану операцію з ініціалізацією / обчисленням результату =).

Операції == та != у Kotlin застосовні для порівняння аргументів довільних типів. Зокрема, дозволяється порівнювати на рівність рядки – вони рівні, якщо мають рівну довжину, і відповідні їх символи збігаються: `"abc" != "cba"`.

Решта операції застосовні тільки в тому випадку, якщо для аргументів визначена спеціальна функція порівняння `compareTo` – детальніше така функція розглядатиметься в наступних роботах. Наразі, нам доведеться застосовувати їх тільки для числових типів.

Математично результат усіх операцій порівняння належить до типу `Boolean`, що має рівно два можливих значення: `true, false`.

Таблична форма розгалужень (when)

Каскадний запис if..else if..else часто можна представити більш витончено в табличній формі, використовуючи конструкцію when (коли). Конструкція when складається з послідовності записів вигляду умова -> результат. В останньому запису умова замінюється на ключове слово else (інакше).

Для прикладу `quadraticRootNumber` це робиться так:

```
fun quadraticRootNumber(a: Double, b: Double, c: Double): Int {
    // Застосовуємо готову функцію з попередньої роботи
    val d = discriminant(a, b, c)
    // Для порівняння на рівність застосовуємо ==
    return when {
        d > 0.0  -> 2
        d == 0.0 -> 1
        else     -> 0
    }
}
```

Частий випадок застосування when – ситуація, коли один і той самий вираз необхідно послідовно порівняти на

рівність з кількома іншими. Для прикладу, розглянемо задачу формування словесної нотації для оцінки. Згідно з існуючими домовленостями, оцінка «5» записується як «відмінно», «4» як «добре», «3» як «задовільно» і «2» як «незадовільно». Представимо подібне перетворення у вигляді функції мовою Kotlin, використовуючи when:

```
fun gradeNotation(grade: Int): String = when (grade) {
    5 -> "відмінно"
    4 -> "добре"
    3 -> "задовільно"
    2 -> "незадовільно"
    else -> "неіснуюча оцінка $grade"
}
```

Ця функція приймає на вхід цілочисельну оцінку (grade) та формує на виході відповідний до неї рядок. Рядки в Kotlin мають тип `String` та записуються в подвійних лапках.

Для перевірки можливого значення `grade` ми використовуємо конструкцію `when (grade)`, у якій воно послідовно порівнюється з 5, 4, 3 та 2. Зверніть увагу, що в нашому запису `when` є ще і п'ятий випадок (else). Його присутність необхідна, оскільки функція повинна знати, який результат їй треба повертати на вихід, для будь-якого припустимого значення входу (у цьому випадку це тип `Int` з його діапазоном допустимих значень). Строго кажучи, гілка `else` тут відповідає помилковій ситуації, яка може передбачати спеціальне опрацювання – але про це пізніше. У функції `gradeNotation` у такій ситуації ми формуємо рядок «неіснуюча оцінка», дописуючи до неї значення переданої оцінки, наприклад: «неіснуюча оцінка 0».

Логічні функції та операції

Умова в операторі `if` часто обчислюється за допомогою функції з результатом типу `Boolean`. Нехай, наприклад, є коло на площині з центром у точці (x_0, y_0) і радіусом r, а також точка на площині з координатами (x, y). Необхідно

визначити, чи лежить точка всередині кола. Особливість даного завдання в тому, що в нього є тільки дві відповіді: ТАК чи НІ або більш формально, ІСТИННО (true) або ХИБНЕ (false).

Для вирішення даного завдання необхідно скористатися нерівністю кола

$$(x - x_0)^2 + (y - y_0)^2 \le r^2.$$

Якщо точка (x, y) задовольняє цій нерівності, то вона лежить усередині кола, якщо ж ні, то вона знаходиться зовні. Функція дуже проста й записується так:

```
fun pointInsideCircle(
        x: Double, y: Double,
        x0: Double, y0: Double, r: Double
    ) = sqr(x - x0) + sqr(y - y0) <= sqr(r)
```

Тут знову використовується функція `sqr` з попередньої роботи для обчислення квадратів чисел. Тип результату функції `pointInsideCircleBoolean`. Під час написання тестових функцій для неї зручно використовувати готові функції `assertTrue` і `assertFalse`, наприклад:

```
@Test
fun pointInsideCircle() {
   // (1, 1) inside circle: center = (0, 0), r = 2
    assertTrue(pointInsideCircle(1.0, 1.0, 0.0, 0.0, 2.0))
   // (2, 2) NOT inside circle: center = (0, 0), r = 2
  assertFalse(pointInsideCircle(2.0, 2.0, 0.0, 0.0, 2.0))
}
```

Обидві функції мають один параметр типу `Boolean`: `assertTrue` (перевірити на істину) призводить до невдалого результату тесту, якщо її аргумент дорівнює **false**, та продовжує виконання тесту, якщо він дорівнює **true,** `assertFalse` (перевірити на хибність) працює з точністю до навпаки.

Функцію `pointInsideCircle` зазвичай можна використовувати для розв'язання більш складних задач. Для поєднання умов можна використовувати логічні операції:

`&&` – логічне І, результат дорівнює **true**, якщо ОБИДВА аргументи **true**;

`||` – логічне АБО, результат дорівнює **true**, якщо ХОЧА Б ОДИН з аргументів дорівнює **true;**

`!` – логічне НЕ, результат дорівнює **true**, якщо аргумент **false.**

Використовуючи логічні операції, можна формулювати комбіновані умови. Наприклад, умова приналежності точки **перетину** або **об'єднанню** двох кіл може виглядати так:

```
// Фрагмент програми...
val x = 0.5
val y = 0.5
// Перетин: логічне І
if (pointInsideCircle(x, y, 0.0, 0.0, 1.0) &&
    pointInsideCircle(x, y, 1.0, 1.0, 1.0)) {
    //...
}
// Об'єднання: логічне АБО
if (pointInsideCircle(x, y, 0.0, 0.0, 1.0) ||
    pointInsideCircle(x, y, 1.0, 1.0, 1.0)) {
    //...
}
// Не належить
if (!pointInsideCircle(x, y, 0.0, 0.0, 1.0)) {
    //...
}
```

Лабораторний практикум № 3

Мета: набути практичних навичок з розробки програм, що реалізують алгоритми з розгалуженням.

Завдання

Завдання 3.1

Надати математичний запис фрагмента програми та обчислити значення змінної x після його виконання (табл. 3.1).

Позначення: *n – номер варіанта.*

Таблиця 3.1.

Варіанти	Фрагмент

1–2	`val t = 17 * n`
	`var x = t`
	`if (t < 10 || t > 30) x=3`
	`else if (t <= 20) x = 0`

3–4	`val t = n`
	`var x = 0`
	`if (t < 0) x = -t`
	`else x = t`

5–6	`val a = n`
	`val b = 13`
	`val c = 12`
	`var x = a`
	`if (x < b) x = b`
	`if (x < c) x = c`

7–8	`val a = n`
	`val b = 17`
	`val c = 18`
	`var x = a`
	`if (b < x) x = b`
	`if (c < x) x = c`

9–10	`val t = n`
	`var x = 0`
	`if (t > 10) x = t * t-n`
	`if (t < 10) x = t`

11–12	`val t = n`
	`var x = t % 4`
	`if (t > 1 && t < 3) x = t`
	`if (t <=1) x = 1`

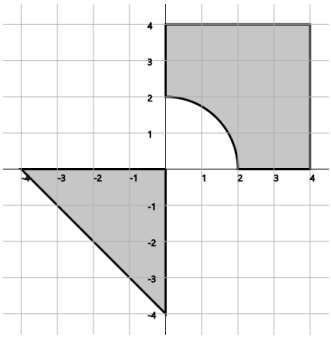
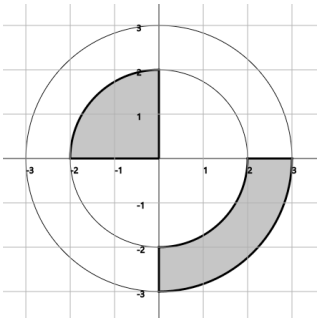
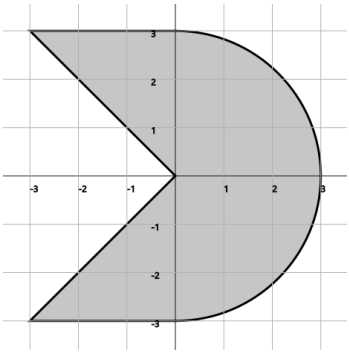
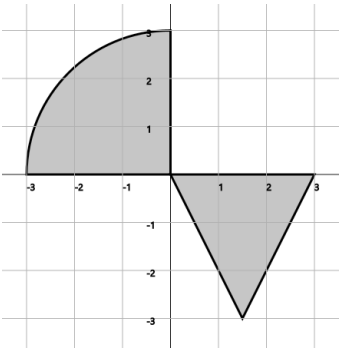
13–14	`val t = n`
	`var x = t`
	`if (t > 0 && t < 10) x=1`
	`if (t >= 0)`
	`  x = 1 / (exp(t) - 1)`

15–16	`var x = -7`
	`val t = x.pow(n)`
	`if (t > 0)`
	`  x = t.pow(1.0 / 3)`
	`else x = t * t * t`

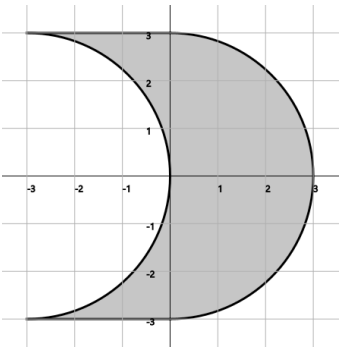
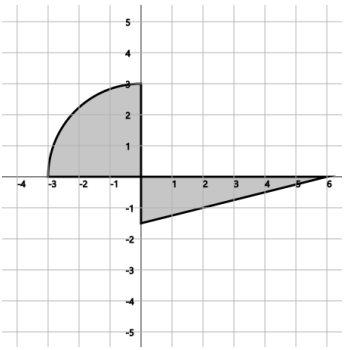
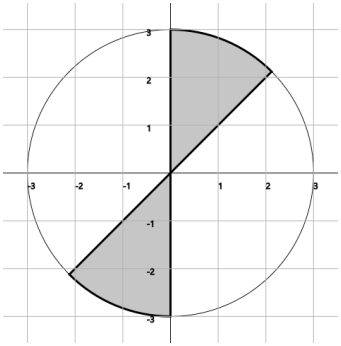
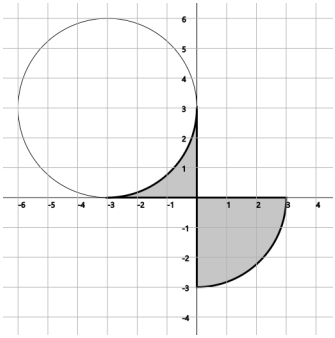
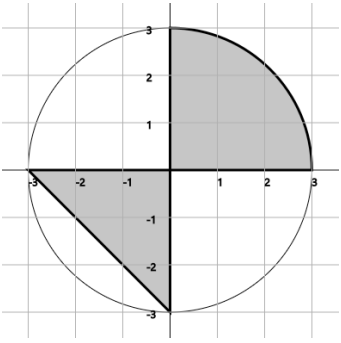
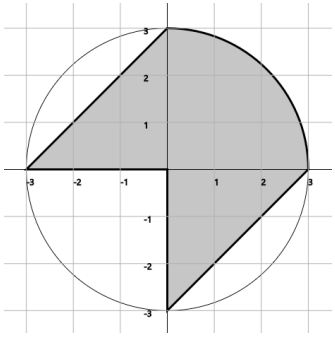
Завдання 3.2

Написати функцію, що виводить на екран значення **true**, якщо точка A з координатами x, y належить до заштрихованої області, та **false** в іншому випадку (табл. 3.2) [6].

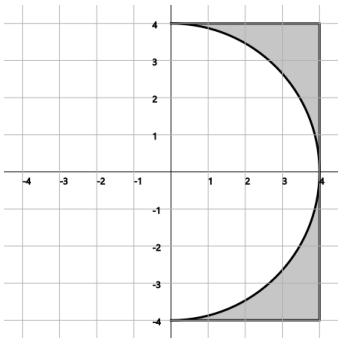
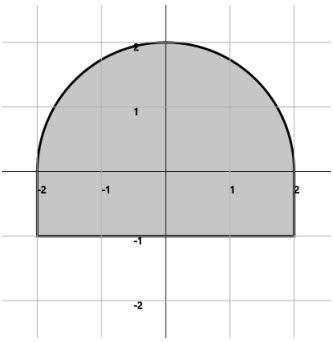
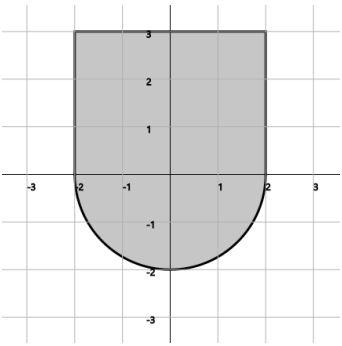
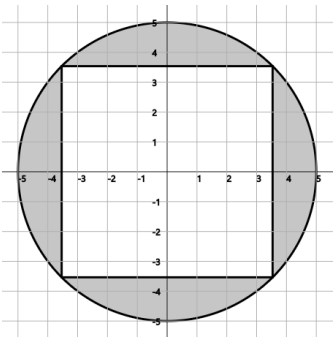
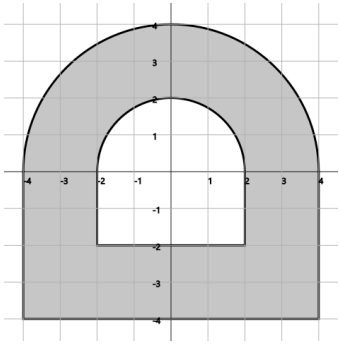
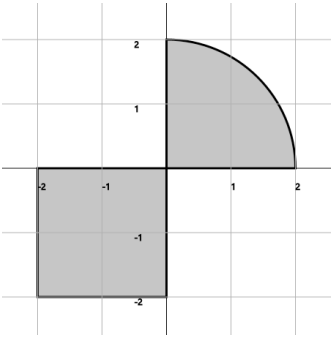
Таблиця 3.2.

Варіант 1	Варіант 2
	
Варіант 3	**Варіант 4**
	

Продовж. табл. 3.2

Варіант 5	Варіант 6
	
Варіант 7	**Варіант 8**
	
Варіант 9	**Варіант 10**
	

Продовж. табл. 3.2

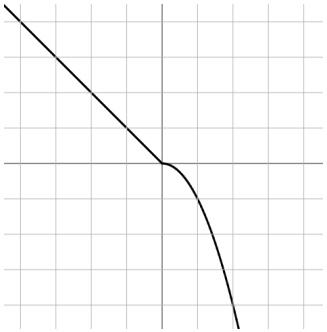
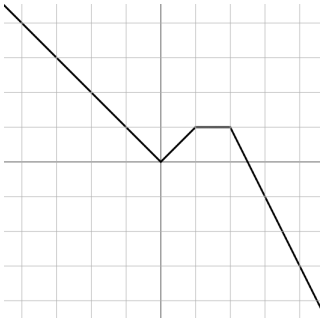
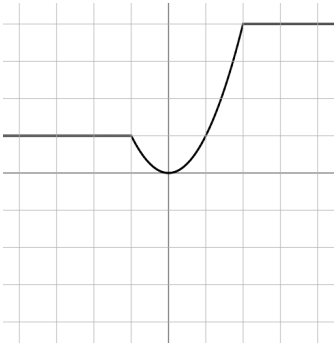
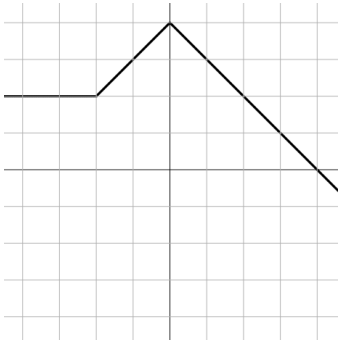
Варіант 11	Варіант 12
	
Варіант 13	**Варіант 14**
	
Варіант 15	**Варіант 16**
	

Завдання 3.3

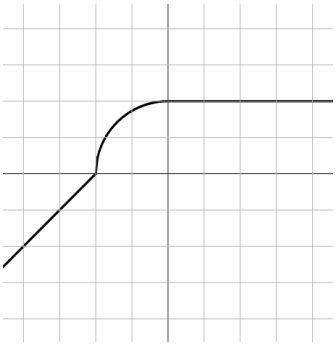
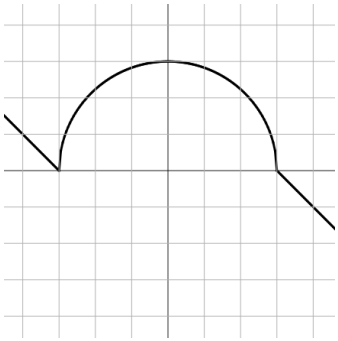
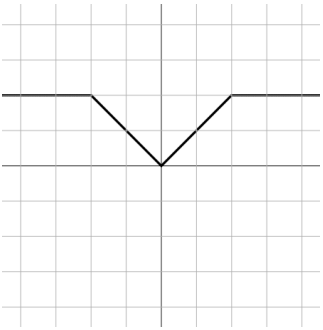
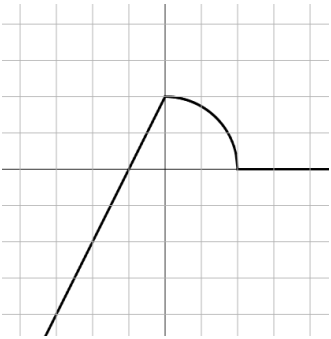
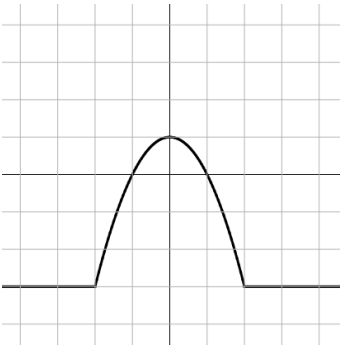
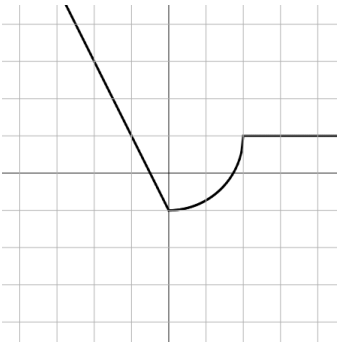
Скласти програму, що містить опис функції, яка задана графічно. Доповнити програму функцією main, яка викликає створену функцію для значення аргументу x, що треба вводити з клавіатури, та виводить результат обчислень на консоль.

Примітка. Одна клітина на графіку дорівнює одиниці.

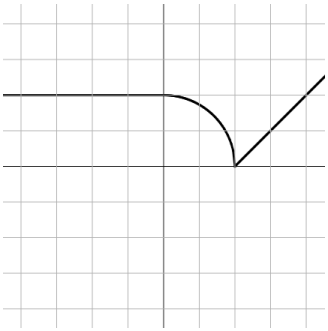
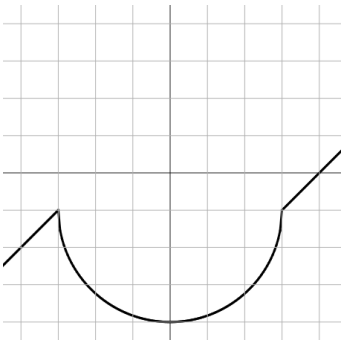
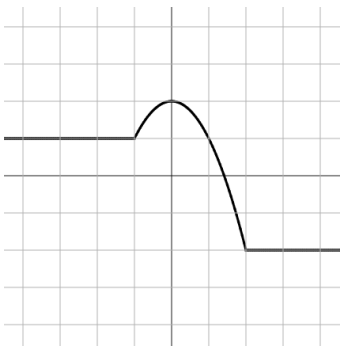
Таблиця 3.3.

Варіант 1	Варіант 2
	
Варіант 3	**Варіант 4**
	

Продовж. табл. 3.3

Варіант 5	Варіант 6
	
Варіант 7	**Варіант 8**
	
Варіант 9	**Варіант 10**
	

Продовж. табл. 3.3

Варіант 11	Варіант 12
	
Варіант 13	**Варіант 14**
	
Варіант 15	**Варіант 16**
	

Контрольні питання

1. Для чого потрібні оператори розгалуження?
2. Які форми операторів розгалуження є в Kotlin?
3. Наведіть приклади оператора if у формі виразу.
4. Наведіть приклади оператора when у формі виразу.
5. Наведіть приклади блочних операторів розгалуження.
6. Як записуються складні умови?
7. Як обчислюються логічні вирази в Kotlin?

Тема 4. ЦИКЛИ ТА РЕКУРСІЇ

Майже всі реальні завдання вимагають для свого рішення не тільки розгалужень, але й багаторазового повторення однотипних дій. Простим прикладом такого завдання є обчислення факторіала `factorial(n) = n!`. Нагадаємо, що факторіал натурального числа n дорівнює добутку всіх чисел від 1 до n. За прийнятою угодою `0! = 1` та `1! = 1`, факторіал від негативного числа не визначений.

Найбільш простим для побудови програмної реалізації є індуктивне визначення факторіала, згідно з яким `n! = n(n-1)!`. Базою індуктивного визначення слугують угоди про значення 0! та 1!. Мовою Kotlin подібне визначення реалізується в такий спосіб:

```
fun factorial(n: Int) = if (n < 2) 1 else n * factorial(n - 1)
```

Тут було застосовано прийом програмування під назвою рекурсія. Для обчислення результату функції `factorial` відбувається виклик цієї ж функції, але з меншим значенням аргументу. Рано чи пізно умова `if (n<2)` виявиться істинною, і значення функції буде обчислено.

Цикл for, змінні, інтервали та прогресії

Можлива також реалізація ітеративного визначення, а саме: `n! = 1 * 2 * … * (n-1) * n`. Для цього необхідно використовувати цикли; найбільш поширеним у Kotlin є цикл `for`, який підходить і для цього завдання. Реалізація буде виглядати так:

```
fun factorial(n: Int): Int {
    // Змінна, що мутує (var)
    var result = 1
    for (i in 1..n) {
        result = result * i
    }
    return result
}
```

Конструкцію `for (i in 1..n) {…}` слід читати як «Для всіх `i` в інтервалі від 1 до n (виконати)…». У даній конструкції оголошується параметр циклу `for`, якому надається ім'я `i`. Усередині фігурних дужок знаходиться тіло циклу `for`. Тіло циклу в даному випадку буде виконано n разів, тобто відбудеться n ітерацій. На кожній ітерації параметр `i` матиме різні значення – від 1 до n.

Рядком вище оголошена так звана мутуюча змінна result. Для її оголошення ми використовували ключове слово `var` (variable) – на відміну від `val` (value) для звичайних змінних. Мутуюча змінна отримує значення 1 при її оголошенні, але в процесі виконання функції вона може змінювати своє значення.

Оператор `result = result * i` виконує так зване присвоювання змінній, що мутує, іншого значення. При цьому спочатку береться її поточне значення (наприклад, 2 на 3-й ітерації циклу), воно множиться на значення параметра i (3 на 3-й ітерації циклу) і результат (6) присвоюється змінній result. Якби ми оголосили змінну `result` як `val result = 1`, то в цьому рядку функції ми отримали б помилку:

```
val cannot be reassigned
```

В останньому операторі return result визначається остаточний результат обчислення факторіала.

Оператор `for` може використовуватися НЕ тільки для перебору чисел в інтервалі (від меншого до більшого

з кроком 1), але і для перебору чисел у заданій прогресії. Інтервал, як вже було показано раніше, створюється оператором виду `min..max`. Типові приклади створення прогресії виглядають так:

• `10 downTo 1` – прогресія від більшого числа до меншого, з кроком 1 (10, 9, 8, …, 1);

• `1..99 step 2` – прогресія від меншого числа до більшого, але з кроком 2 (у цьому випадку, перебір усіх непарних чисел за зростанням);

• `100 downTo 2 step 2` – прогресія від більшого числа до меншого, з кроком 2 (перебір усіх парних чисел за спаданням).

Модифікуючі оператори

Відкрийте файл MainLab4.kt з проєкту за посиланням https://github.com/kafedra-iust/lab4kotlin в IDE і подивіться на визначення функції factorial зверху файла. Можна побачити, що оператор `result = result * i` підкреслений сірою хвилястою лінією. Якщо навести на нього курсор миші, можна побачити повідомлення

`Replace with *= operator` ,

тобто "Замінити оператором *=". Після натискання Alt + Enter з'явиться контекстне меню з символом лампочки й командою "Replace with *= operator". Натиснувши Enter, у IDE виконається заміна, що запропонована.

Новий текст функції буде таким:

```
fun factorial(n: Int): Int {
    // Мутуюча змінна (var)
    var result = 1
    for (i in 1..n) {
        result *= i
    }
    return result
}
```

Оператор `*=` відноситься до великої групи модифікуючих операторів і виконує домноження поточного значення змінної result на значення параметра i. Аналогічно йому працюють оператори `+=`, `-=`, `/=`, `%=` та деякі інші.

Послідовна перевірка умов

Розглянемо тепер дещо складніший приклад. Нехай нам потрібно написати функцію, яка перевіряє натуральне число на простоту. Нагадаємо, що число називається простим (prime), якщо воно ділиться без остачі тільки на одиницю і на себе, і складеним, якщо в нього є й інші дільники (одиниця зазвичай не вважається ні простим, ні складеним числом).

Прямолінійна перевірка передбачає ділення заданого числа n послідовно на числа в інтервалі від 2 до n-1. Щоб перевірити, чи ділиться число n націло на інше число m, досить порівняти залишок від ділення n % m з нулем. Якщо хоча б раз ми успішно поділили без остачі – початкове число n не є простим.

```
fun isPrime(n: Int): Boolean {
    if (n < 2) return false // оскільки 1 - не просте число
    for (m in 2..n - 1) {
        if (n % m == 0) return false
    }
    return true
}
```

Зверніть увагу, що, знайшовши дільник, ми відразу повідомляємо про те, що результат – false – при цьому перериваєтся як виконання циклу, так і виконання функції, оскільки результат вже визначений. Щоб довести, що число є складеним, нам достатньо знайти хоча б ОДИН дільник від 2 до n-1. Однак про результат true ми можемо повідомити тільки після закінчення циклу, перевіривши ВСІ дільники: щоб довести простоту числа, треба переконатися,

що ЖОДНЕ число від 2 до n-1 не є дільником. Початківці часто роблять ось таку помилку:

```
for (m in 2..n - 1) {
       if (n % m == 0) return false
       else return true
}
```

що, звісно, неправильно. Такий цикл буде виконано тільки один раз, результат буде true для непарних і false для парних чисел.

У тестовій функції можна перевірити, що числа 2, 5 і 11 є простими, а числа 1, 4, 9 і 15 – складеними. Також можна написати більш складну перевірку, використовуючи той факт, що перші 1000 простих чисел лежать в інтервалі від 2 до 7919.

```
@Test
fun isPrime() {
    var count = 0
    for (n in 2..7919) {
        if (isPrime(n)) {
            count++
        }
    }
    assertEquals(1000, count)
}
```

У циклі перевіряються числа від 2 до 7919 на простоту. Кожен раз, коли число виявляється простим, виконується оператор `count++` – скорочена форма запису `count = count + 1` або `count += 1`, так званий оператор інкремента (існує також оператор `--`, або оператор декремента).

Спробуємо тепер за допомогою `isPrime` дізнатися, скільки існує простих чисел, менших десяти мільйонів (для цього достатньо замінити в наведеному фрагменті коду 7919 на 10000000). Якщо запустити таку функцію на виконання, воно триватиме досить багато часу. Причина в тому, що наша функція `isPrime (n: Int)` виконує зайві

перевірки. Зокрема, досить перевірити ділимість числа n на всі числа в інтервалі від 2 до n/2, оскільки на великі числа n усе одно ділитися не буде. Більш того, достатньо обмежитись інтервалом від 2 до $\sqrt{x}$ – якщо n ділиться на деяке більше $\sqrt{x}$ число (наприклад, 50 ділиться на 10), то воно буде ділитись і на деяке менше число (у цьому випадку, 50 ділиться на 5 = 50/10).

```
fun isPrime(n: Int): Boolean {
    if (n < 2) return false // оскільки 1 -- не просте
число
    for (m in 2..sqrt(n.toDouble()).toInt()) {
        if (n % m == 0) return false
    }
    return true
}
```

Зверніть увагу, що перед обчисленням квадратного кореня ми були змушені скористатися функцією `n.toDouble()` для отримання дійсного числа з цілого, а після обчислення – функцією `.toInt()` для отримання цілого числа з дійсного. Обидві ці вбудовані в Kotlin функції мають незвичайну для початківців форму запису, яку слід читати як "n перетворити до Double", "…перетворити до Int". Замість того, щоб записати аргумент усередині круглих дужок `toDouble(n)`, ми записуємо його перед ім'ям функції, відокремлюючи його від імені символом точки. Подібний аргумент функції називається її отримувачем (receiver), у майбутньому така форма запису буде використовуватися неодноразово.

Переривання та продовження циклу

Під час програмування циклів часто зустрічаються ситуації, коли необхідно перервати виконання циклу достроково, або ж достроково перейти до початку його наступної ітерації. З цією метою в мові Kotlin використовуються оператори `break` і `continue`.

Продемонструємо їх на прикладі. Досконалим числом називається таке натуральне число, що дорівнює сумі всіх своїх дільників, крім себе самого. Зокрема, 6 = 1 + 2 + 3 і 28 = 1 + 2 + 4 + 7 + 14 – досконалі числа. Напишемо функцію, що визначає, чи є задане число n досконалим.

```
fun isPerfect(n: Int): Boolean {
    var sum = 1
    for (m in 2..n/2) {
        if (n % m == 0) {
            sum += m
            if (sum > n) break
        }
    }
    return sum == n
}
```

Ця функція перебирає всі можливі дільники числа n від 2 до n / 2 (Одиницю перебирати безглуздо, оскільки на неї ділиться будь-яке число – тому мутуюча змінна sum спочатку дорівнює 1, а не 0). Кожен знайдений дільник додається до суми. Якщо в якийсь момент набрана сума виявилася більшою n – цикл можна перервати за допомогою `break`, оскільки наступні подільники можуть тільки збільшити її ще більше. Після переривання циклу виконується наступний за ним оператор, у цьому випадку return.

Інший варіант запису тієї ж самої функції, який використовує оператор продовження continue:

```
fun isPerfect(n: Int): Boolean {
    var sum = 1
    for (m in 2..n/2) {
        if (n % m > 0) continue
        sum += m
        if (sum > n) break
    }
    return sum == n
}
```

Тут замість того, щоб перевірити, що n ділиться на m, ми перевіряємо зворотну умову – що n НЕ ДІЛИТЬСЯ на m. Якщо умова істина, виконується оператор continue, при цьому залишок даної ітерації циклу пропускається, відбувається збільшення значення m на 1 і перехід до наступної ітерації. Обидві реалізації isPerfect рівнозначні, застосування тієї чи іншої з них – справа смаку.

Цикли while і do..while

Іноді трапляється так, що необхідний цикл не зводиться до перебору якогось заздалегідь відомого набору елементів. У цьому випадку в мові Kotlin замість циклу for застосовуються цикли while або do..while. Як приклад розглянемо таку задачу: знайти число входжень цифри m (від 0 до 9) у десятковий запис невід'ємного числа n. Наприклад, у число n = 5373393 цифра m = 3 входить чотири рази.

Для вирішення цього завдання нам необхідно в циклі перебрати всі цифри числа n. Для отримання молодшої цифри числа досить взяти залишок від його ділення на 10, для відкидання молодшої цифри, слід розділити його на 10. За допомогою циклу while це записується в такий спосіб:

```
fun digitCountInNumber(n: Int, m: Int): Int {
    var count = 0
    var number = n
    while (number > 0) {
        if (m == number % 10) {
            count++
        }
        number /= 10
    }
    return count
}
```

На відміну від циклу for, цикл while потенційно може виконатися будь-яку кількість разів. Перед кожною новою ітерацією циклу (у тому числі перед першою), цикл while

перевіряє записану в дужках умову. Якщо вона правильна, ітерація виконується, якщо – ні, цикл завершується. Для даного прикладу, при n = 5373393 виконається сім ітерацій циклу – відповідно до кількості цифр у числі.

Якщо уважно порозмірковувати, можна побачити, що така реалізація «зламається» для наведеного далі тесту

```
@Test
fun digitCountInNumber() {
    assertEquals(1, digitCountInNumber(0, 0))
}
```

У цьому прикладі ми очікуємо, що цифра 0 входить у число 0 один раз. Однак, написана вище функція дає відповідь 0. Виправити функцію можна таким чином:

```
fun digitCountInNumber(n: Int, m: Int): Int {
    var count = 0
    var number = n
    do {
        if (m == number % 10) {
            count++
        }
        number /= 10
    } while (number > 0)
    return count
}
```

У цьому прикладі цикл while було замінено циклом do..while. Відмінність його полягає в тому, що умова після ключового слова while перевіряється не ПЕРЕД кожною ітерацією, а ПІСЛЯ кожної ітерації, через це тіло циклу do..while завжди виконується хоча б один раз. Тому дані цикли називаються *циклом з передумовою* (*while*) або *циклом з післяумовою* (*do..while*).

Конкретно для випадку з n = 0 цикл while не буде виконано жодного разу, і результат залишиться таким, що дорівнює 0. Цикл do..while буде виконаний один раз, у числі буде знайдена цифра 0 і результат буде дорівнювати 1, тобто в даному конкретному випадку цикл do..while краще підхо-

дить для розв'язання задачі. У загальному випадку, будь-яке завдання може бути вирішене із застосуванням будь-якого з цих двох циклів, питання лише в тому, яке рішення буде виглядати краще. Цикл while на практиці зустрічається значно частіше.

Зауважимо, що у даного завдання можливо і рекурсивне рішення. Як його можна придумати? Для цього спочатку слід розв'язати задачу у тривіальному випадку – для n < 10. До того ж результат буде дорівнювати 1, якщо m = n, і 0 в іншому випадку. Після цього слід придумати перехід від числа з великою кількістю цифр до числа або чисел, у яких цифр менше. Наприклад, число n можна розбити на два інших: n%10, що містить тільки останню цифру, та n/10, що містить усі інші цифри:

```
fun digitCountInNumber(n: Int, m: Int): Int =
    when {
        n == m -> 1
        n < 10 -> 0
           else -> digitCountInNumber(n / 10, m) +
                digitCountInNumber(n % 10, m)
}
```

Зверніть увагу, що рекурсивне рішення часто коротше й витонченіше ітеративного.

Лабораторний практикум № 4

Мета: набути практичних навичок розробки циклічних програм та основ створення рекурсивних функцій.

Завдання

Завдання 4.1

Надати математичний запис фрагмента програми та обчислити значення змінної x після його виконання, де n – це номер варіанта.

Таблиця 4.1.

Варіанти	Фрагмент

1–3	`var x = 1`
	`for (j in 7 downTo n) x *= j`
	`x *= 2`

4–6	`var x = 0`
	`var j = 1`
	`do {`
	`  x += j`
	`  j += 2`
	`} while (j <=n)`

7–9	`var x = 0.0`
	`var k - 3 * n`
	`while (k > 0) {`
	`  x = sqrt(k+x)`
	`  k -= 3`
	`}`

10–12	`var x = n`
	`for (k in 0..5) {`
	`  if (k < 2) continue`
	`  x++`
	`}`

13–15	`var x = 0`
	`for (j in 0 until n)`
	`x += 2`
	`x *= 2`

16–18	`var x = 1`
	`while (x <= n)`
	`x++ x *= 2`

Завдання 4.2

Скласти програму табулювання функції f(x) на відрізку [a; b] з кроком h. Значення a, b, h вводити з клавіатури.

Таблиця 4.2.

Варіант	Функція	Варіант	Функція
1	$y = \frac{tgx}{lnx}$	2	$y = \sqrt[3]{x}$

Продовж. табл. 4.2

Варіант	Функція	Варіант	Функція
3	$y = tg(lnx)$	4	$y = \frac{ln(x-1)}{4-x}$
5	$y = tg^2(lnx)$	6	$y = ctg(lnx)$
7	$y = x^{0.2}$	8	$y = \frac{x}{1+tgx}$
9	$y = \frac{ln(x-0.5)}{\sqrt{x}}$	10	$y = \frac{sinx^3}{2-x}$
11	$y = \frac{cos^3 x}{1-lgx}$	12	$y = \frac{tgx}{lnx-1}$
13	$y = \frac{e^2}{\sqrt{1-x^2}}$	14	$y = \frac{x+1}{\sqrt{x}} \quad -\sqrt[4]{\lvert x-2 \rvert}$
15	$y = \frac{ln\lvert 2x \rvert}{sinx-\pi}$	16	$y = e^{lnx-1} + sinx$
17	$y = \frac{4-x}{ln(x-1)}$	18	$y = \frac{e^{x^2}}{\sqrt{1-x^2}}$

Завдання 4.3

Для заданих *x*, *n*, *eps*, що вводяться з клавіатури:

a. Обчислити *n* доданків згідно з варіантом.

b. Обчислити суму тих доданків, які за абсолютним значенням більше *eps*. (Завдання виконати для двох різних *eps*, які відрізняються на порядок, для кожного випадку обчислити кількість доданків).

c. Порівняти результати з «точним» значенням відповідної функції (сума визначає наближене значення) для x Є (-R,R).

Примітка 1. У цьому завданні значення x повинно належати області допустимих значень, визначеної в пункті c, значення n повинно бути достатньо великим (більше 20).

eps потрібно обирати як маленьке додатне число, яке не більше 1e-9. Виконати обчислення для двох РІЗНИХ eps, що відрізняються не менше ніж на порядок (у 10 або більше разів).

Примітка 2. Результат виконання пункту b вважається задовільним, та програма може бути зарахована як правильна, лише за умови, що різниця між «точним» значенням функції та значеннями, обчисленими за наближеними сумами, взята за модулем, є числом, що менше або дорівнює eps.

Таблиця 4.3.

Варіант	Завдання
1	$\frac{sinx}{x}=1-\frac{x^2}{3!}+\frac{x^4}{5!}-\frac{x^6}{7!}+\ldots\left(R=\infty\right)$
2	$e^{-x^2}=1-\frac{x^2}{1!}+\frac{x^4}{2!}-\ldots+\left(-1\right)^n\frac{x^{2n}}{n!}\left(R=\infty\right)$
3	$ln\left(x+\sqrt{x^2+1}\right)=x-\frac{1}{2}\cdot\frac{x^3}{3}+\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{x^5}{5}-\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\frac{x^7}{7}+\ldots\left(R=1\right)$
4	$arctgx=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\frac{x^9}{9}-\ldots\left(R=1\right)$
5	$arcsinx=x+\frac{1}{2}\cdot\frac{x^3}{3}+\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{x^5}{5}+\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\frac{x^7}{7}+\ldots\left(R=1\right)$
6	$\frac{1}{\sqrt{1-x^2}}=1+\frac{1}{2}\cdot x^2+\frac{1}{2}\cdot\frac{3}{4}\cdot x^4+\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot x^6+\ldots\left(R=1\right)$
7	$\frac{1}{\sqrt{1+x}}=1-\frac{1}{2}\cdot x+\frac{1}{2}\cdot\frac{3}{4}\cdot x^2-\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot x^3+\ldots\left(R=1\right)$
8	$\sqrt{1+x}=1+\frac{1}{2}\cdot x-\frac{1}{2\cdot4}\cdot x^2+\frac{1\cdot3}{2\cdot4\cdot6}\cdot x^3-\ldots\left(R=1\right)$
9	$\frac{1}{\left(1+x\right)^3}=1-\frac{2\cdot3}{2}\cdot x+\frac{3\cdot4}{2}\cdot x^2-\frac{4\cdot5}{2}\cdot x^3+\ldots\left(R=1\right)$
10	$\frac{1}{\left(1+x\right)^2}=1-2x+3x^2-4x^3+5x^4-\ldots\left(R=1\right)$
11	$\frac{1}{1+x}=1-x+x^2-x^3+x^4-\ldots\left(R=1\right)$

Продовж. табл. 4.3

Варіант	Завдання
12	$ln\frac{1+x}{1-x}=2\cdot\left(x+\frac{x^3}{3}+\frac{x^5}{5}+\frac{x^7}{7}+\frac{x^9}{9}+\ldots\right)(R=1)$
13	$ln(1-x)=-\frac{x}{1}-\frac{x^2}{2}-\frac{x^3}{3}-\frac{x^4}{4}-\ldots(R=1)$
14	$ln(1+x)=\frac{x}{1}-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}-\ldots(R=1)$
15	$ch(x)=1+\frac{x^2}{2!}+\frac{x^4}{4!}+\frac{x^6}{6!}+\ldots(R=\infty)$
16	$sh(x)=1+\frac{x^3}{3!}+\frac{x^5}{5!}+\frac{x^7}{7!}+\ldots(R=\infty)$
17	$cos(x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots(R=\infty)$
18	$sin(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots(R=\infty)$
19	$e^x=1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots(R=\infty)$
20	$e^{-x}=1-\frac{x}{1!}+\frac{x^2}{2!}-\frac{x^3}{3!}+\ldots(R=\infty)$

Контрольні питання

1. Що таке інтервал? Як він утворюється?
2. Для чого потрібні інтервали та прогресії в Kotlin?
3. Які прогресії бувають?
4. Які види циклів є в Kotlin?
5. Наведіть приклад циклу for.
6. Наведіть приклади циклів while та do…while.
7. Для чого існують оператори переривання та продовження циклів?
8. Що таке рекурсія? Для чого потрібні рекурсивні функції?

Тема 5. МАСИВИ ТА СПИСКИ

Списки

З цієї роботи ми починаємо вивчення складених типів даних, що складаються з декількох елементів простих типів. Такі типи дуже часто необхідні в програмуванні. Розглянемо, наприклад, задачу на пошук усіх коренів біквадратного рівняння

$$ax^4 + bx^2 + c = 0.$$

Чи можна написати функцію, яка цю задачу вирішить? Звичайно, так, але результатом подібної функції повинен бути список знайдених коренів біквадратного рівняння. Список – це один з дуже поширених складених типів з наступними властивостями:

- Список може включати в себе будь-яку кількість елементів (від нуля до нескінченності).
- Кількість елементів у списку називається його розміром.
- Усі елементи списку мають один і той же тип (зі свого боку цей тип може бути простим – список дійсних чисел, або складеним – список рядків, або список списків цілих чисел, або будь-які інші варіанти).
- В іншому елементи списку незалежні один від одного.

Розглянемо розв'язання задачі на пошук коренів біквадратного рівняння мовою Kotlin:

```
fun biRoots(a: Double, b: Double, c: Double):
List<Double> {
    if (a == 0.0) {
        if (b == 0.0) return listOf()
        val bc = -c / b
        if (bc < 0.0) return listOf()
        val root = sqrt(bc)
        return if (root == 0.0) listOf(root)
               else listOf(-root, root)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    val y1 = (-b + sqrt(d)) / (2 * a)
    val y2 = (-b - sqrt(d)) / (2 * a)

// part1: List<Double>
    val part1 =
    if (y1 < 0) listOf() else if (y1 == 0.0)
listOf(0.0) else {
        val x1 = sqrt(y1)
        listOf(-x1, x1)
     }
// part2: List<Double>
    val part2 =
    if (y2 < 0) listOf() else if (y2 == 0.0)
listOf(0.0) else {
        val x2 = sqrt(y2)
        listOf(-x2, x2)
     }
    return part1 + part2
}
```

Цей розв'язок побудовано за алгоритмом, що описаний нижче:

1. Перший if розглядає тривіальний випадок `a = 0` і більш просте рівняння $bx^2 = -c$. Воно або не має коренів `(c / b > 0)`, або має один корінь, що дорівнює нулю `(c / b = 0)`, або два кореня `(c / b < 0)`.

2. Потім ми робимо заміну $y = x^2$ та обчислюємо дискримінант $d = b^2 - 4ac$. Якщо він від'ємний, рівняння не має коренів.

3. Якщо дискримінант дорівнює 0, рівняння $ay^2 + by + c = 0$ має один корінь. Залежно від його знаку, біквадратне рівняння або не має коренів, або має один корінь 0, або має два кореня.

4. В іншому випадку дискримінант додатний і рівняння $ay^2 + by + c = 0$ має два корені. Кожен з них, у залежності від його знаку, перетворюється в нуль, один або два корені біквадратних рівняння.

Розглянемо тип результату функції biRoots – він зазначений як `List<Double>`. List у Kotlin – це і є список. У кутових дужках <> вказується так званий типовий аргумент – тип елементів списку, тобто `List<Double>` разом – це список дійсних чисел.

Для створення списків, зручно використовувати функцію `listOf()`. Аргументи цієї функції – це елементи створюваного списку, їх може бути будь-яка кількість (у тому числі 0). У ряді випадків, коли біквадратне рівняння не має коренів, функція biRoots повертає порожній список результатів.

В останньому, найскладнішому випадку, коли рівняння $ay^2 + by + c = 0$ має два корені `y1` і `y2`, ми формуємо рішення рівнянь $x^2 = y1$ і $x^2 = y2$ у вигляді списків `part1` і `part2`. Обидві ці проміжні змінні мають тип `List<Double>` – у цьому можна переконатися в IDE, поставивши на них курсор введення і натиснувши комбінацію клавіш Ctrl + Q. В останньому операторі `return` ми складаємо два цих списки один з одним: `return part1 + part2`, утворюючи таким чином третій список, що містить у собі всі елементи двох попередніх.

Функцію biRoots можна дещо спростити, звернувши увагу на те, що ми в ній чотири рази розв'язуємо одну й ту ж задачу: пошук коренів рівняння $x^2 = y$. Для програміста така ситуація повинна відразу перетворюватися в сигнал – для вирішення цього завдання слід написати окрему, більш просту функцію:

```
fun sqRoots(y: Double) =
        if (y < 0) listOf()
        else if (y == 0.0) listOf(0.0)
        else {
            val root = sqrt(y)
            // Результат!
            listOf(-root, root)
        }
```

Якщо уважно подивитися на оператор `if..else if.. else`, можна побачити, що перші дві його гілки формують результат відразу ж, використовуючи `listOf()` та `listOf(0.0)`. А ось гілка `else` спочатку створює проміжну змінну `root` і вже потім формує результат `listOf(-root, root)`.

Важливо! *Результат гілки в таких випадках формує останній її оператор.*

Цю ж функцію можна переписати з використанням оператора `when`:

```
fun sqRoots(y: Double) =
        when {
            y < 0 -> listOf()
            y == 0.0 -> listOf(0.0)
            else -> {
                val root = sqrt(y)
                // Результат!
                listOf(-root, root)
            }
        }
```

З використанням sqRoots функція biRoots набуде наступного вигляду:

```
fun biRoots(a: Double, b: Double, c: Double): List<Double> {
    if (a == 0.0) {
        return if (b == 0.0) listOf()
        else sqRoots(-c / b)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    if (d == 0.0) return sqRoots(-b / (2 * a))
    val y1 = (-b + sqrt(d)) / (2 * a)
    val y2 = (-b - sqrt(d)) / (2 * a)
    return sqRoots(y1) + sqRoots(y2)
}
```

З вихідних 24 рядків залишилося тільки 11, та й розуміння тексту функції стало значно простіше.

Напишемо тепер тестову функцію для перевірки роботи функції biRoots. З цією метою послідовно вирішимо з її допомогою такі рівняння:

$0x^4 + 0x^2 + 1 = 0$ (коренів немає);
$0x^4 + 1x^2 + 2 = 0$ (коренів немає);
$0x^4 + 1x^2 - 4 = 0$ (корені –2, 2);
$1x^4 - 2x^2 + 4 = 0$ (коренів немає);
$1x^4 - 2x^2 + 1 = 0$ (корені –1, 1);
$1x^4 + 3x^2 + 2 = 0$ (коренів немає);
$1x^4 - 5x^2 + 4 = 0$ (корені –2, –1, 1, 2).

```
fun biRootsTest() {
   assertEquals(listOf<Double>(), biRoots(0.0, 0.0, 1.0))
   assertEquals(listOf<Double>(), biRoots(0.0, 1.0, 2.0))
   assertEquals(listOf(-2.0, 2.0), biRoots(0.0, 1.0, -4.0))
   assertEquals(listOf<Double>(), biRoots(1.0, -2.0, 4.0))
   assertEquals(listOf(-1.0, 1.0), biRoots(1.0, -2.0, 1.0))
   assertEquals(listOf<Double>(), biRoots(1.0, 3.0, 2.0))
   assertEquals(listOf(-2.0, -1.0, 1.0, 2.0),
                biRoots(1.0, -5.0, 4.0))
}
```

Зверніть увагу, що тут ми використовуємо запис `listOf<Double>()` для створення порожнього списку. Справа в тому, що для викликів на кшталт `listOf(-2.0, 2.0)`, тип елементів створюваного списку зрозумілий з аргументів функції – це `List<Double>`. А ось виклик `listOf()` без аргументів не дає ніякої інформації про тип елементів списку, у той же час, наприклад, порожній список рядків і порожній список цілих чисел – з точки зору мови Kotlin – різні типи даних.

У багатьох випадках Kotlin, тим не менш, може зрозуміти, про який список іде мова. Наприклад, функція `biRoots` має результат `List<Double>`, тому, усі списки, які використовуються в операторах `return`, повинні мати такий же тип. Випадок з викликом `assertEquals`, однак, не несе достатньої інформації, щоб зрозуміти тип елементів, і ми змушені записати виклик функції більш детально – `listOf<Double>()`, указуючи типовий аргумент `<Double>` між ім'ям функції, що викликається і списком її аргументів у круглих дужках.

Запустимо тепер написану тестову функцію. Ми отримаємо провалений тест через останню перевірку:

```
org.opentest4j.AssertionFailedError: expected: <[-2.0,
-1.0, 1.0, 2.0]> but was: <[-2.0, 2.0, -1.0, 1.0]>
```

Тобто ми очікували список коренів –2, –1, 1, 2, а отримали натомість –2, 2, –1, 1. Справа в тому, що списки в Kotlin вважаються рівними, якщо збігаються їх розміри, і відповідні елементи списків дорівнюють один одному. Списки, що складаються з одних і тих же елементів, але на різних місцях, вважаються різними.

У цьому місці програміст повинен задуматися, а що, власне, він хоче в точності від функції `biRoots`. Чи повинні знайдені корені бути впорядковані за зростанням, або вони можуть бути присутніми у списку в будь-якому поряд-

ку? Якщо повинні, то він має виправити функцію `biRoots`, а якщо ні – то тестову функцію, оскільки вона вимагає від тестуємої функції більше, ніж та за фактом дає.

В обох випадках нам доведеться впорядкувати список знайдених коренів перед порівнянням. У мові Kotlin це можна зробити, викликавши функцію `.sorted ()`:

```
fun biRootsTest() {
    assertEquals(listOf(-2.0, -1.0, 1.0, 2.0),
                 biRoots(1.0, -5.0, 4.0).sorted())
}
```

Раніше ми вже зустрічалися з функціями з одержувачем `.toInt()` і `.toDouble()`. Функція `.sorted()` також вимагає наявності одержувача: виклик `list.sorted()` створює список того ж розміру, та з тих самих елементів, що й вихідний, але його елементи будуть впорядковані за зростанням.

Поширені операції над списками

Перерахуємо деякі операції над списками, що присутні в бібліотеці мови Kotlin:

1. `listOf(…)` – створення нового списку.
2. `list1 + list2` – додавання двох списків, сума списків містить усі елементи їх обох.
3. `list + element` – додавання списку та елемента, сума містить усі елементи list і додатково element.
4. `list.size` – отримання розміру списку (Int).
5. `list.isEmpty(), list.isNotEmpty()` – отримання ознак порожнечі й непорожнечі списку (Boolean).
6. `list[i]` – індексація, тобто отримання елемента списку з цілочисельним індексом (номером) i. За правилами мови Kotlin, у списку з n елементів вони мають індекси, що починаються з нуля: 0, 1, 2, …, останній елемент списку має індекс `n-1`. Тобто, під час використання запису `list[i]` повинно бути справедливо `i >= 0 && i < list.size`. Інакше

виконання програми буде перерване з помилкою (використання індексу за межами списку).

7. `list.sublist(from, to)` – створення списку меншого розміру (підсписку), у який увійдуть елементи списку `list` з індексами `from, from + 1, …, to-2, to-1`. Елемент з індексом `to` не включається.

8. `element in list` – перевірка приналежності елемента `element` списку `list`.

9. `for (element in list) { … }` – цикл `for`, що перебирає всі елементи списку `list`.

10. `list.first()` – отримання першого елемента списку (якщо список порожній, виконання програми буде перервано з помилкою).

11. `list.last()` – отримання останнього елемента списку (аналогічно).

12. `list.indexOf(element)` – пошук індексу елемента `element` у списку `list`. Результат цієї функції дорівнює `-1`, якщо елемент у списку відсутній. Інакше, звертаючись до списку `list` по обчисленому індексу, ми отримаємо `element`.

13. `list.min(), list.max()` – пошук мінімального й максимального елемента у списку.

14. `list.sum()` – сума елементів у списку.

15. `list.sorted(), list.sortedDescending()` – побудова відсортованого списку (за зростанням чи за спаданням) з поточного списку.

16. `list1 == list2` – порівняння двох списків на рівність. Списки рівні, якщо співпадають їхні розміри та відповідні елементи.

Змінні списки

Змінний список є різновидом звичайного, його тип визначається як `MutableList<ElementType>`. На додаток до тих можливостей, які є у всіх списках у мові Kotlin, змінний

список може змінюватися по ходу виконання програми або функції. Це означає, що змінний список дозволяє:

1. Змінювати свій уміст операторами `list[i] = element`.
2. Додавати елементи в кінець списку, зі збільшенням розміру на 1: `list.add(element)`.
3. Видаляти елементи зі списку, зі зменшенням розміру на 1 (якщо елемент був у списку): `list.remove(element)`.
4. Видаляти елементи зі списку по індексу, зі зменшенням розміру на 1: `list.removeAt(index)`.
5. Вставляти елементи в середину списку: `list.add(index, element)` – вставляє елемент `element` по індексу `index`, зі зсувом усіх наступних елементів на 1, наприклад `listOf(1, 2, 3).add(1, 7)` дасть результат `[1, 7, 2, 3]`.
6. Для створення змінного списку можна використовувати функцію `mutableListOf(…)`, аналогічну `listOf(…)`.

Розглянемо приклад. Нехай є вихідний список цілих чисел `list`. Потрібно побудувати список, що складається з його від'ємних елементів, порядок їх у списку повинен залишитися незмінним. Для цього потрібно:

- Створити порожній змінний список.
- Пройтися по всіх елементах вихідного списку й додати їх у змінний список, якщо вони від'ємні.
- Повернути заповнений змінний список.

```
fun negativeList(list: List<Int>): List<Int> {
    val result = mutableListOf<Int>()
    for (element in list) {
        if (element < 0) {
            result.add(element)
        }
    }
    return result
}
```

Тут проміжна змінна `result` має тип `MutableList<Int>` (переконатися в цьому можна в IDE за допомогою комбіна-

ції Ctrl + Q). Незважаючи на це, її можна використовувати в операторі `return` функції з результатом `List<Int>`. Відбувається це тому, що тип `MutableList<Int>` є різновидом типу `List<Int>`, тобто будь-який змінний список є також і просто списком (зворотне неправильно – не кожний список є мутуючим). Мовою математики це означає, що ОДЗ (область допустимих значень) типу `MutableList<Int>` є підмножиною ОДЗ типу `List<Int>`.

У наступному прикладі функція приймає на вхід уже змінний список цілих чисел, і змінює в ньому всі додатні числа на протилежні за знаком:

```
fun invertPositives(list: MutableList<Int>) {
    for (i in 0 until list.size) {
        val element = list[i]
        if (element > 0) {
            list[i] = -element
        }
    }
}
```

Використана тут функція `list.withIndex()` з вихідного списку формує інший список, містить пари (індекс, елемент), а цикл `for ((index, element) in …)` перебирає паралельно елементи та їхні індекси. Що таке пара, і як нею користуватися в Kotlin, докладніше буде розглянуто в наступних роботах.

У загальному й цілому, нечасто варто користуватися функціями, основний сенс яких полягає у зміні їхніх параметрів. Наприклад, так виглядає тестова функція для `invertPositives`:

```
fun invertPositivesTest() {
    val list1 = mutableListOf(1, 2, 3)
    invertPositives(list1)
    assertEquals(listOf(-1, -2, -3), list1)
    val list2 = mutableListOf(-1, 2, 4, -5)
    invertPositives(list2)
    assertEquals(listOf(-1, -2, -4, -5), list2)
}
```

Якщо раніше одна перевірка завжди займала один рядок, то в цьому прикладі вона займає три рядки, через необхідність створення проміжних змінних `list1` і `list2`. Крім цього, факт зміни `list1`, `list2` під час виклику `invertPositives` схильний вислизати від уваги читача, ускладнюючи розуміння програми.

Масиви

Масив Array – ще один тип, призначений для зберігання і модифікації деякої кількості однотипних елементів. З точки зору можливостей, масив схожий на змінний список `MutableList`; головною його відмінністю є відсутність можливості змінювати свій розмір – для масивів відсутні функції `add` та `remove`.

Для звернення до елемента масиву слугує оператор індексації: `array[i]`, причому є можливість як читати вміст масиву, так і змінювати його. Для створення масиву, зручно використовувати функцію `arrayOf()`, аналогічну `listOf()` для списків.

Майже всі можливості, наявні для списків, є і для масивів теж. Винятком є функції для створення підсписків `sublist`. Також масиви не слід порівнювати на рівність за допомогою `array1 == array2`, оскільки в багатьох випадках таке порівняння дає неправильний результат. Масив можна перетворити до звичайного списку за допомогою `array.toList()` або до змінного списку за допомогою `array.toMutableList()`. Список так само можна перетворити до масиву за допомогою `list.toTypedArray()`.

Загалом під час написання програм мовою Kotlin, майже немає випадків, коли необхідно використовувати саме масиви. Одним з небагатьох прикладів є головна функція, параметр якої має тип `Array<String>` – через нього в програму передаються аргументи командного рядка. Але в нових версіях мови Kotlin цей параметр не є обов'язковим.

Лабораторний практикум № 5

Мета: набути практичних навичок розробки програм, що використовують списки та/або масиви.

Завдання

Завдання 5.1

Надати математичний запис фрагмента програми з табл. 5.1 та обчислити значення змінної x після його виконання. Елементи масиву обчислюються за формулою

```
a[i+1] = (67 * a[i] + 11) % 128.
```

Значення a[0] дорівнює номеру варіанта за списком групи.

Таблиця 5.1.

Варіанти	Фрагмент програми

1–3	`val t = 2`
	`val n = 3`
	`var x = 0`
	`for (j in 0 until n) {`
	`  x = x * t + a[j] }`

4–6	`val n = 4`
	`var x = a[n]`
	`for (j in n-1 downTo 0) {`
	`  x = a[j] + 1 / x }`

7–9	`val n = 4`
	`var x = a[0]`
	`for (j in 1 until n) {`
	`  if (a[j] < x) x = a[j]`
	`}`

10–12	`val t = 3.0`
	`val n = 3`
	`var x = a[n].toDouble()`
	`for (j in 0 until n) {`
	`  x = x + a[j] * t.pow(n-j)`
	`}`

Продовж. табл. 5.1

Варіанти	Фрагмент програми

13–15	`val n = 4`
	`val m - n / 2`
	`var k = n - 1`
	`for (j in 0 until m) {`
	`  val y = a[j]`
	`  a[j] = a[k]`
	`  a[k] = y`
	`  k--`
	`}`
	`val x = a[0]`

16–18	`val n = 4`
	`var k = 0`
	`var x = 0.0`
	`for (j in 0 until n) {`
	`  if (a[j] > 0) {`
	`  x += a[j]`
	`  k++`
	`  }`
	`}`
	`if (k!=0 x /= k)`

Завдання 5.2

a. Протабулювати функцію з таблиці 4.2. Значення `x` та `y` занести у списки.

b. Знайти найбільше та найменше значення у списку `y`. Вивести їх та відповідні їм значення з масиву `x` у наступному вигляді:

```
yMin = … при x = …
yMax = … при x = …
```

c. Обчислити суму та середнє арифметичне значення елементів масиву y. Результати вивести на екран.

Завдання 5.3

Скласти програму обчислення величин з табл. 5.2 та виконати її у середовищі програмування. Елементи списку (масиву) визначаються за формулою

`a[i] = p[i] – 64;` де `p[i+1] = (p[i] * 67 + 11) % 128.`

де p[0] дорівнює N – номеру варіанта за списком групи, кількість елементів у списку дорівнює 50.

Таблиця 5.2.

Варіант	Завдання
1	Найбільший елемент масиву a та його порядковий номер
2	Сума елементів масиву a, значення яких кратні N
3	Сума елементів масиву a, значення яких парні числа
4	Середнє арифметичне додатних елементів масиву a
5	Сума елементів масиву a, значення яких непарні числа
6	Середнє геометричне додатних елементів масиву a
7	Сума елементів масиву a, значення яких двозначні непарні числа
8	Добуток найбільшого та найменшого елементів масиву a
9	Сума елементів масиву a, значення яких двозначні парні числа
10	Модуль вектора a/3
11	Найменший елемент масиву a, з парним номером
12	Найбільший непарний елемент масиву
13	Сума від'ємних елементів масиву
14	Сума квадратів елементів масиву з парними номерами
15	Середнє арифметичне найбільшого та найменшого елементів з парними номерами
16	Середнє арифметичне найбільшого парного та найменшого непарного елементів

Контрольні питання

1. Для чого потрібні масиви?
2. Для чого потрібні списки?
3. Які списки існують у Kotlin?
4. Що таке ініціалізатор масиву?
5. Які операції підтримують масиви та списки?

Тема 6. ДВОВИМІРНІ МАСИВИ ТА РЯДКИ

Двовимірні масиви

Часто одного масиву недостатньо. У деяких випадках зручно використовувати двовимірні масиви. Візуально їх легко уявити у вигляді сітки. Типовий приклад – зал у кінотеатрі. Щоб знайти потрібне місце в залі кінотеатру, нам потрібно знати ряд і місце.

Двовимірний масив – це масив, який містить інші масиви.

Розглянемо приклад. Створимо двовимірний масив 5x5 і заповнимо його нулями.

```
// Створюємо двовимірний масив
var matrix = arrayOf<Array<Int>>()

// заповнюємо нулями
for (i in 0..4) {
    var array = arrayOf<Int>()
    for (j in 0..4) {
        array += 0
    }
    matrix += array
}

// виводимо дані масиву
for (array in cinema) {
    for (value in array) {
        print("$value ")
```

```
    }
   println()
}
```

У результаті отримаємо такий масив, що буде виведений на екран

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Якщо розміри масиву відомі заздалегідь, можна створити масив і одразу заповнити його нулями (чи будь-якими іншими значеннями):

```
val n = 5
val matrix = Array(n){ Array(n) { 0 } }
```

Зауважимо, що іноді, коли елементами одновимірного масиву є примітивні значення (з мови Java), для підвищення швидкодії програми можна скористатися спеціальними типами масивів: `IntArray`, `LongArray`, `DoubleArray`, `CharArray` та ін. Наприклад, попередній фрагмент можна переписати так:

```
val n = 5
val matrix = Array(n){ IntArray(n) { 0 } }
```

Нагадаємо також, що в мові Kotlin дуже часто замість масивів використовуються списки. При цьому треба мати на увазі, що списки List є незмінними, тобто значення, які будуть записані у відповідні елементи під час ініціалізації, змінити буде неможливо. Списки також можуть бути двовимірними:

```
val n = 5
val matrixOf0 = List(n){ List(n) { 0 } }
val mutableMatrix = List(n){ MutableList(n) {0} }
```

Також можна комбінувати масиви і списки, наприклад, створити список масивів:

```
val n = 5 val matrix = List(n){ IntArray(n) { 0 } }
```

Примітка. *У подальшому там, де використовується термін «двовимірний масив», мається на увазі одна з показаних вище конструкцій.*

Для отримання доступу до елемента двовимірного масиву, треба вказати два індекси: індекс у зовнішньому масиві (найчастіше, це – номер рядка) та індекс внутрішнього масиву (найчастіше, це – номер стовпця).

Наприклад:

```
matrix[2][1] = 100 // елемента з індексами 2,1 присвоюємо значення 100 // заповнення діагональних елементів матриці одиницями
for (i in matrix.indices) {
    for (j in matrix[i].indices) {
        matrix[i][j] = 1
    }
}
```

Розглянемо більш складний приклад. Створимо масив, заповнюючи елементи, що розташовуються на головній діагоналі одиницями, а інші – нулями. Звісно, для цього можна було скористатися кодами, наведеними в попередніх прикладах, проте існує більш компактний спосіб зробити це, використовуючи ініціалізатори:

```
// змінна n - розмір матриці, визначена раніше
val matrix = Array(n){
   row -> IntArray(n) { col -> if (row == col) 1 else 0 }
}
```

Рядки

У мові Kotlin рядки – це об’єкти, що складаються з послідовності символів, мають відповідну довжину (кількість символів) та методи для роботи з ними.

Для визначення довжини рядка існує змінна length:

```
val s = "Hello, World!"
val len = s.length // 13
```

Також існують деякі властивості – «розширення», наприклад, `CharSequence.indices: IntRange` – визначен-

ня діапазону індексів рядка, та `CharSequence.lastIndex: Int` – визначення індексу останнього елемента рядка.

Як і в мові Java, у Kotlin рядки є незмінними об'єктами. Це означає, що будь-яка операція з рядком, не може змінити його та, за потреби, може створити новий рядок. Більшість операцій з рядками є функціями з «отримувачем» – тобто викликаються із синтаксисом "рядок".операція(). Наведемо таблицю основних рядкових функцій мови Kotlin (табл. 6.1).

Таблиця 6.1. **Основні рядкові функції мови Kotlin**

Функція	Опис
`fun compareTo(other: String): Int`	Порівняти рядок з іншим у лексикографічному порядку
`fun equals(other: Any?): Boolean`	Порівняти, чи співпадає рядок з іншим (тобто чи складаються вони з однакових символів, розташованих в однаковому порядку)
`fun get(index: Int): Char`	Отримати символ рядка, що стоїть на позиції index (починаючи з 0)
`operator fun plus(other: Any?): String`	Приєднати до рядка рядкове представлення об'єкта
`fun subSequence (startIndex: Int, endIndex: Int): CharSequence`	Отримати підрядок, що починається з startIndex та завершується перед endIndex
`fun CharSequence. all(predicate: (Char) → Boolean): Boolean`	Перевірка, чи всі символи рядка задовольняють умову
`inline fun CharSequence. any(predicate: (Char) → Boolean): Boolean`	Перевірка, чи є в рядку хоча б один символ, що задовольняє умову
`fun CharSequence. commonPrefixWith(other: CharSequence, ignoreCase: Boolean = false): String`	Визначити загальний префікс рядка з іншим рядком

Продовж. табл. 6.1

Функція	Опис
`fun CharSequence.commonSuffixWith(other: CharSequence, ignoreCase: Boolean = false): String`	Визначити загальний суфікс рядка з іншим рядком
`fun String.concat(str: String): String`	Поєднати рядки
`operator fun CharSequence.contains(other: CharSequence, ignoreCase: Boolean = false): Boolean`	Перевірка, чи містить рядок інший рядок
`operator fun CharSequence.contains(char: Char, ignoreCase: Boolean = false): Boolean`	Перевірка, чи містить рядок указаний символ
`operator fun CharSequence.contains(regex: Regex): Boolean`	Перевірка, чи містить рядок послідовність символів, визначену регулярним виразом
`fun CharSequence.count(): Int`	Визначити кількість символів у рядку
`inline fun CharSequence.count(predicate: (Char) → Boolean): Int`	Визначити кількість символів у рядку, що задовольняють указану умову
`fun CharSequence.drop(n: Int): CharSequence`	Видалити перші n символів рядку
`fun CharSequence.last(): Char`	Отримати останній символ рядку
`inline fun CharSequence.dropWhile(predicate: (Char) → Boolean): CharSequence`	Видалити всі перші символи рядка, що задовольняють умову

Продовж. табл. 6.1

Функція	Опис
`fun CharSequence.dropLast(n: Int): CharSequence`	Видалити n останніх символів рядка
`inline fun CharSequence.dropLastWhile(predicate: (Char) → Boolean): CharSequence`	Видалити всі останні символи рядка, що задовольняють умову
`inline fun CharSequence.elementAtOrElse(index: Int, defaultValue: (Int) → Char): Char`	Отримати символ рядка, що стоїть у вказаній позиції, або defaultValue, якщо такої позиції в рядку немає
`fun CharSequence.elementAtOrNull(index: Int): Char?`	Отримати символ рядка, що стоїть у вказаній позиції, або null, якщо такої позиції в рядку немає
`fun String.endsWith(suffix: String, ignoreCase: Boolean = false): Boolean`	Перевірка, чи закінчується рядок указаною послідовністю символів
`inline fun CharSequence.filter(predicate: (Char) → Boolean): CharSequence`	Сформувати новий рядок, відібравши з нього лише ті символи, що задовольняють умову
`inline fun CharSequence.filterNot( predicate: (Char) → Boolean ): CharSequence`	Сформувати новий рядок, відібравши з нього лише ті символи, що не задовольняють умову
`fun CharSequence.first(): Char`	Визначити перший символ рядка
`inline fun CharSequence.first( predicate: (Char) → Boolean ): Char`	Визначити перший символ рядка, що задовольняє умову
`inline fun CharSequence.forEach(action: (Char) → Unit)`	Виконати вказану дію для кожного символу в рядку
`fun String.format(vararg args: Any?): String`	Відформатувати рядок

Продовж. табл. 6.1

Функція	Опис
`fun CharSequence. indexOf( char: Char, startIndex: Int = 0, ignoreCase: Boolean = false ): Int`	Знайти індекс символу в рядку
`fun CharSequence. isEmpty(): Boolean`	Перевірити, чи є рядок порожнім
`fun CharSequence. isNotEmpty(): Boolean`	Перевірити, чи не є рядок порожнім
`fun CharSequence?. isNullOrEmpty(): Boolean`	Перевірити, чи є рядок порожнім або відсутнім
`fun String.lowercase(): String`	Перевести символи рядка в нижній регістр
`fun String. removePrefix(prefix: CharSequence): String`	Видалити з рядка вказаний префікс, якщо він є
`fun String.removeRange( startIndex: Int, endIndex: Int ): String`	Видалити символи рядка, що знаходяться в межах указаного діапазону
`fun String. removeSuffix(suffix: CharSequence): String`	Видалити з рядка вказаний суфікс, якщо він є
`fun String.remove Surrounding( prefix: CharSequence, suffix: CharSequence ): String`	Видалити з рядка вказані префікс та суфікс, якщо вони є
`fun String.replace( oldChar: Char, newChar: Char, ignoreCase: Boolean = false ): String`	Замінити в рядку всі символи oldChar на newChar
`fun String.replace( oldValue: String, newValue: String, ignoreCase: Boolean = false ): String`	Замінити в рядку всі фрагменти oldValue на newValue

Продовж. табл. 6.1

Функція	Опис
`fun String.reversed(): String`	Отримати «обернений» рядок, що складається з символів вихідного рядка, записаних у зворотному порядку
`fun CharSequence.split( regex: Regex, limit: Int = 0 ): List<String>`	Розділити рядок на список рядків, використовуючи вказаний regex як роздільник
`fun String.startsWith( prefix: String, ignoreCase: Boolean ): Boolean`	Перевірити, чи починається рядок з указаного префікса
`fun String. substring(startIndex: Int): String`	Перевірити, чи містить рядок указаний суфікс
`fun String.trim(): String`	«Обрізати» рядок, видаливши з нього всі початкові та завершальні пробільні символи
`fun String.uppercase(): String`	Перетворити рядок до верхнього регістру

Лабораторний практикум № 6

Мета: набути практичних навичок розробки програм з використанням двовимірних масивів та рядків за допомогою засобів мови Kotlin.

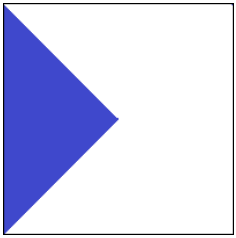
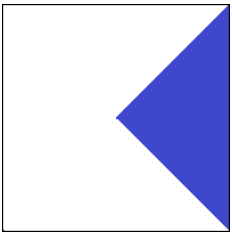
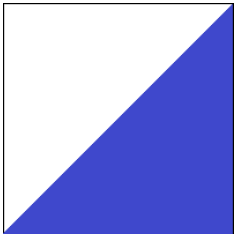
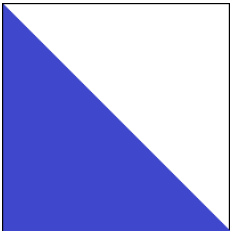
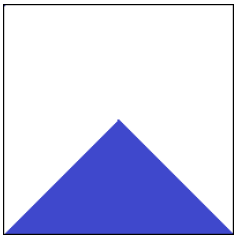
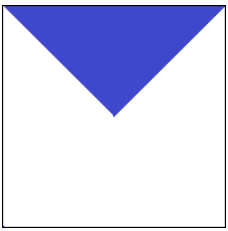
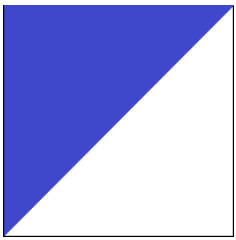
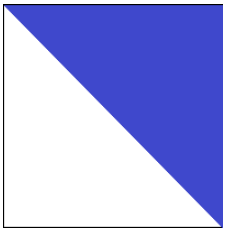
Завдання

Завдання 6.1

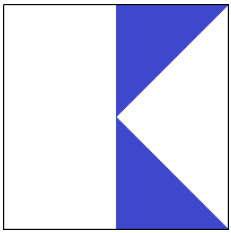
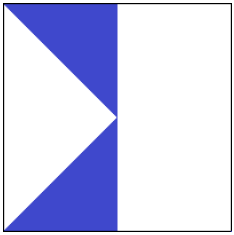
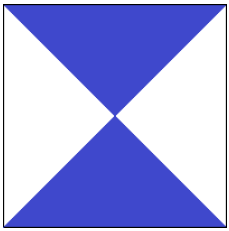
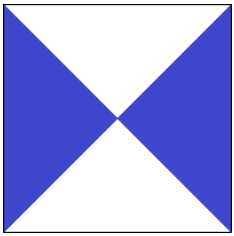
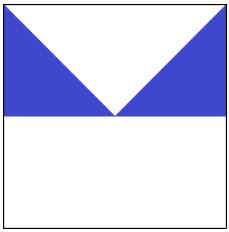
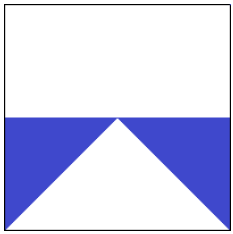
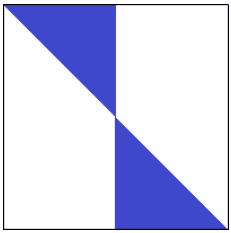
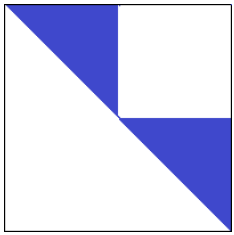
Заповнити двовимірний масив згідно з правилом заданого рисунком у табл. 6.2. Після заповнення, масив обов'язково вивести на екран.

Примітка. Розмір масиву вводити з клавіатури. Усі елементи, що потрапляють до зафарбованої області дорівнюють 1, усі інші – дорівнюють 0.

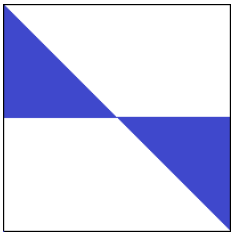
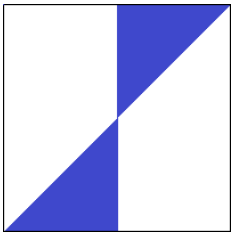
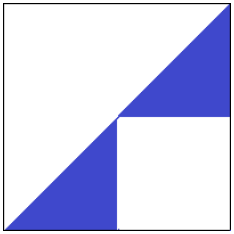
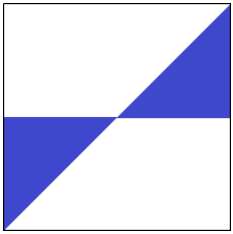
Таблиця 6.2.

Варіант 1	Варіант 2
	
Варіант 3	Варіант 4
	
Варіант 5	Варіант 6
	
Варіант 7	Варіант 8
	

Продовж. табл. 6.2

Варіант 9	Варіант 10
	
Варіант 11	Варіант 12
	
Варіант 13	Варіант 14
	
Варіант 15	Варіант 16
	

Продовж. табл. 6.2

Варіант 17	Варіант 18
	
Варіант 19	Варіант 20
	

Завдання 6.2

Згідно з варіантом із завдання 6.1 знайти суму та середнє арифметичне значення елементів зафарбованої частини масиву. Результати обчислень вивести на екран.

Примітка. Для виконання завдання попередньо заповнити масив випадковими цілими числами з проміжку від –50 до 50.

Завдання 6.3

Скласти програму для виконання дій за табл. 6.3 й виконати її в середовищі програмування.

Примітка. У всіх завданнях вважати, що рядок може містити лише літери, цифри та знаки пробілу. Слово – послідовність символів, що не містить пробілів.

Таблиця 6.3.

Варіанти	Завдання
1–3	Визначення кількості слів у рядку
4–6	Вилучення всіх цифр у рядку
7–9	Інвертування символів у рядку
10–12	Визначення кількост цифр у рядку
13–15	Визначення слова з найменшою кількістю літер у рядку
16–18	Визначення кількості чисел у рядку
19–21	Визначення слова з найбільшою кількістю літер у рядку
22–24	Заміна всіх великих букв у рядку на малі
25–27	Вилучення зайвих символів «пробіл» у рядку (виконати заміну кожної послідовності з двох або більше «пробілів» на один)
28–30	Заміна всіх рядкових (малих) букв у рядку на заголовні (великі)

Контрольні питання

1. Що таке багатовимірний масив?
2. Як описати двовимірний масив у Kotlin?
3. Як ініціалізувати значеннями двовимірний масив у Kotlin?
4. Що таке рядок у Kotlin?
5. Які рядки існують у Kotlin?
6. Наведіть приклади рядкових операцій у Kotlin.

Тема 7. ОСНОВИ МОВИ С/С++

Алфавіт мови С++, ідентифікатори та ключові слова

Для програмування будь-яких завдань мовою С++ необхідно знати такі конструкції: ідентифікатори, службові слова, описи даних та функцій, вирази, оператори, вбудовані функції, управляючі конструкції та деякі інші. Вирази в мові С++ записуються за допомогою 26 рядкових та 26 прописних літер англійського алфавіту: a b c d e f g h i j k I m n o p q r s t u v w x y z, A B C D E F G H I J K L M N O P Q R S T U V W X Y Z; десяти цифр: 0 1 2 3 4 5 6 7 8 9; таких спеціальних символів: + - * / =, . _ : ; ? \ " ’~ | ! # $ & ( ) [ ] { } ^ @ [5].

До спеціальних символів відноситься також пропуск. Комбінації деяких символів, не розділених пропусками, інтерпретуються як один значущий символ: ++ — | | && << >> >= <= == != += -= = /= .?: :: / */ //

Ідентифікаторами називаються імена, що надають змінним, константам, типам даних і функціям, які використовуються в програмах. Після опису ідентифікатора, можна посилатися на об’єкт, що позначається ним [5].

Ідентифікатор – це послідовність символів довільної довжини, яка містить літери, цифри й символи підкрес-

лення, обов'язково починається з літери, або символу підкреслення. У C++ враховується регістр букв. Компілятор сприймає прописні й рядкові букви, як різні символи. Так, змінні `userName` та `UserName` розглядаються як два різні ідентифікатори.

Ключові слова є зарезервованими ідентифікаторами, кожному з яких відповідає певна дія. Змінити призначення ключового слова не можна (директива препроцесора `#define` дозволяє створити «псевдонім» ключового слова, який дублює його дії, можливо, з деякими змінами). Імена ідентифікаторів, що створюються в програмі, не повинні збігатися з ключовими словами мов C/C++ [5].

Таблиця 7.1. **Ключові слова мов C/C++**

`alignas` (починаючи з C++11)	`enum`	`return`
`alignof` (починаючи з C++11)	`explicit`	`short`
`and`	`export`	`signed`
`and_eq`	`extern`	`sizeof`
`asm`	`false`	`static`
`auto`	`float`	`static_assert` (починаючи з C++11)
`bitand`	`for`	`static_cast`
`bitor`	`friend`	`struct`
`bool`	`goto`	`switch`
`break`	`if`	`template`
`case`	`inline`	`this`
`catch`	`int`	`thread_local` (починаючи з C++11)
`char`	`long`	`throw`
`char16_t` (починаючи з C++11)	`mutable`	`true`
`char32_t` (починаючи з C++11)	`namespace`	`try`
`class`	`new`	`typedef`
`compl`	`noexcept` (починаючи з C++11)	`typeid`
`const`	`not`	`typename`
`constexpr` (починаючи з C++11)	`not_eq`	`union`
`const_cast`	`nullptr` (починаючи з C++11)	`unsigned`
`continue`	`operator`	`using`
`decltype` (починаючи з C++11)	`or`	`virtual`
`default`	`or_eq`	`void`
`delete`	`private`	`volatile`
`do`	`protected`	`wchar_t`
`double`	`public`	`while`
`dynamic_cast`	`register`	`xor`
`else`	`reinterpret_cast`	`xor_eq`

Стандартні типи даних

Кожна програма обробляє певну інформацію. У C++ дані мають один з базових типів: **`char`** (текстові дані), **`int`**

(цілі числа), **float** (числа з плаваючою точкою одинарної точності), **double** (числа з плаваючою точкою подвійної точності), **void** (порожні значення), **bool** (логічні значення), перерахування і покажчики.

Текстом (тип даних **char**) є один символ. Зазвичай кожен символ займає 8 біт або один байт з діапазоном значень від 0 до 255.

Цілі числа (тип даних **int**) знаходяться в діапазоні від –32 768 до 32 767 раніше займали 16 біт, тобто два байти, або одне слово. У Windows NT і Windows XP і сучасніших ОС Windows використовуються 32-розрядні цілі числа, що дозволяє розширити діапазон значень від – 2 147 483 648 до 2 147 483 647. У C++ підтримуються три типи цілих чисел. Разом із стандартним типом **int** існують типи **short int** (коротке ціле) і **long int** (довге ціле). Допускається скорочений запис **short** і **long**.

Числа з плаваючою точкою одинарної точності (тип даних **float**) можуть бути подані як у фіксованому форматі, так і в експоненціальному. Діапазон значень – від ± 3.4E–38 до ± 3.4E+38, розмірність – 32 біта, тобто 4 байти або 2 слова. Числа з плаваючою точкою подвійної точності (тип даних **double**) мають діапазон значень від ± 1.7E–308 до ± 1.7E+308 і розмірності 64 біт, тобто 8 байтів або 4 слова. Раніше існував тип **long double** з розмірністю 80 біт і діапазоном від ± 1.18E–4932 до ± 1.18E+4932. У нових 32 розрядних версіях компіляторів він еквівалентний типу **double** й підтримується з міркувань зворотної сумісності з написаними раніше програмами.

Перерахування представляються кінцевим набором іменованих констант різних типів.

Тип даних **void** зазвичай застосовується у функціях, що не повертають ніякого значення. Цей тип даних та-

кож можна використовувати для створення узагальнених покажчиків.

Покажчики, на відміну від змінних інших типів, містять не дані у звичайному розумінні цього слова, а адреси пам'яті, де зберігаються дані.

Змінні нового логічного типу даних **`bool`** у C++ можуть містити тільки одну з двох констант: **`true`** або **`false`**. Компілятор мови C++ дозволяє при описанні змінних указувати модифікатор **`unsigned`**. Він застосовується з чотирма типами даних: **`char`**, **`short`**, **`int`** і **`long`**. Наявність даного модифікатора вказує на те, що значення змінної повинне інтерпретуватися як беззнакове число, тобто найстарший біт є бітом даних, а не бітом знаку. Існує модифікатор **`signed`**, який виконує протилежну **`unsigned`** дію.

Пріоритет операцій

Як і в інших мовах програмування, у мові C++ діють правила пріоритету (порядку виконання) операцій у виразах. Пріоритети операцій наведені в табл. 7.2.

Таблиця 7.2. **Пріоритети операцій**

Операція	Опис	Асоціативність
`++`	Постфіксний (префіксний) інкремент	Зліва направо
`--`	Постфіксний (префіксний) декремент	Зліва направо
`()`	Виклик функції	
`[]`	Доступ до елемента масиву	
`->`	Непрямий доступ до члена класу	
`.`	Прямий доступ до члена класу	
`!`	Логічне НЕ	
`~`	Побітове НЕ	
`-`	Унарний мінус	

Продовж. табл. 7.2

Операція	Опис	Асоціативність
+	Унарний плюс	
&	Отримання адреси	
*	Розкриття покажчика	
sizeof	Отримання розмірності виразу в байтах	
new	Динамічне створення об'єкта	
delete	Динамічне видалення об'єкта	
(тип даних)	Приведення типу	
.*	Прямий доступ до покажчика на член класу (через об'єкт)	Зліва направо
->*	Непрямий доступ до покажчика на член класу (через покажчик на об'єкт)	
*	Множення	Зліва направо
/	Ділення	Зліва направо
%	Ділення по модулю	Зліва направо
+	Додавання	Зліва направо
-	Віднімання	Зліва направо
<<	Зсув ліворуч	Зліва направо
>>	Зсув праворуч	
<	Менше	Зліва направо
>	Більше	
<=	Менше або дорівнює	
>=	Більше або дорівнює	
==	Дорівнює	Зліва направо
!=	Не дорівнює	
&	Побітове І	Зліва направо
^	Побітове АБО (що виключає)	Зліва направо
|	Побітове АБО	Зліва направо
&&	Логічне І	Зліва направо
||	Логічне АБО	Зліва направо

Продовж. табл. 7.2

`?:`	Умовний вираз	Справа наліво
Операція	Опис	Асоціативність
`=`	Просте присвоювання	Справа наліво
`*= /= %= += -= <<= >>= &= |= ^=`	Присвоювання з множенням, діленням …	
`,`	Кома	Зліва направо

У мовах C/C++ усі стандартні функції знаходяться в бібліотеках, які можна підключити за допомогою заголовочних файлів. Так, функції введення-виведення у стилі мови C описані у файлі **`stdio.h (cstdio)`**. Функції та класи для введення-виведення у стилі C++ описані у файлах **`iostream`** (для введення-виведення із використанням стандартного пристрою) та **`fstream`** (для файлового введення-виведення). Обчислення в програмах на C/C++ неможливі без використання математичних функцій, які описані у файлі **`math.h (cmath)`**.

Розглянемо приклад: знаходження значення похідної функції в точці.

Постановка завдання: задана функція $y = sin(x)$. Знайти її похідну в точці $x = \pi / 2$. Для знаходження похідної в точці використовується відомий вираз:

$$y'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

```
#include <math.h>
#include <iostream>
using namespace std;
int main() {
 // Вибираємо приріст аргументу
      double dx=1.0e-11;
```

```
 // Вибираємо точку для обчислення похідної
      double x = 3.1415926;
 // Обчислюване значення функції в точці x+dx
 double f1=sin(x+dx);
 // Обчислюване значення функції в точці x
 double f2=sin(x);
 // Знаходимо значення похідної
      double pf=(f1-f2) /dx;
      cout << "dsin(x) /dx=" << pf <<" x= "<<x;
      return 0;
}
```

Оператор if

Оператор `if` призначений для виконання команди або блоку команд залежно від того, істинно задана умова чи ні.

Формат оператора `if`:

```
if (умова) вираз;
```

Якщо в результаті перевірки умови повертається значення **`true`**, виконується вираз, після чого управління передається наступному рядку програми.

Якщо ж результатом перевірки умови є значення `false`, вираз пропускається. Оператор `if/else` дозволяє вибірково виконувати одну з двох дій залежно від умови. Формат даної інструкції має вигляд:

```
if (умова) вираз 1;
      else вираз 2;
```

Якщо в результаті перевірки умови повертається значення `true`, виконується вираз 1, інакше – вираз 2. Якщо операторна частина гілки `if` або `else` містить не один вираз, а декілька, необхідно укласти їх у фігурні дужки. Після закриваючої фігурної дужки крапка з комою не ставиться. Оператор `if` обох форм реалізують алгоритми, надані на рисунку 7.1.

Як аналізований вираз в операторові `if` найчастіше використовується одна з операцій відношення.

Рис. 7.1. Алгоритми обох форм оператора `if`

Оператор switch/case

Оператор `switch/case` дозволяє залежно від значення деякого виразу вибрати один з багатьох варіантів продовження програми. Оператор має такий формат:

```
switch(expr) {
    case val1: op1; break;
    case val2: op2; break;
...
    case valN: opN; break;
    default: opN+1;
}
```

Оператор `switch/case` реалізує алгоритм, наведений на рис. 7.2.

Оператор `switch/case` може бути використаний у варіанті без оператора N+1. Як вираз при операторі `switch` зазвичай використовується змінна типу `int` або `char`, хоча можна використовувати і складніші вирази, у які входять, наприклад, арифметичні або логічні операції над декількома змінними та константами. Як значення при операторі `case` зазвичай використовуються просто константи (у числовій формі або в символьній, якщо вони були заздалегідь визначені за допомогою оператора препроцесора `#define`), проте можуть використовуватися і вирази над константами. Виконання оператора `switch` починається з обчислення виразу в дужках, який повинен давати цілочисельний результат. Цей результат послідовно порівнюється із значеннями при

операторах `case`, і, якщо буде виявлено рівність результатів, то виконується оператор відповідного `case`. Якщо збіги результатів не виявлено, виконується оператор при операторі `default`, якщо оператор `default` відсутній, то починають виконуватися оператори, наступні за всією конструкцією `switch/case`.

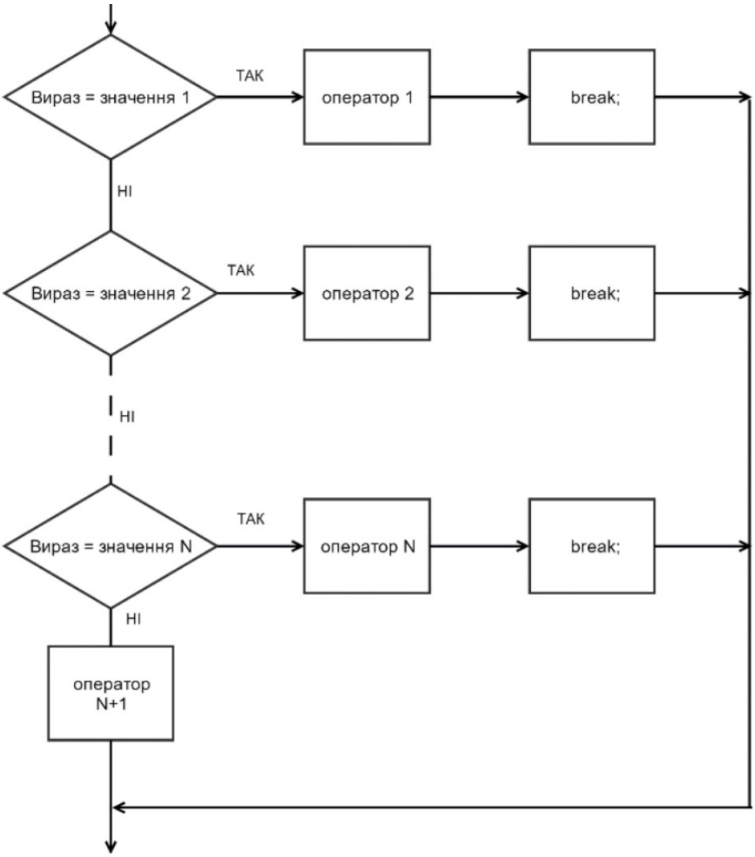


Рис. 7.2. Алгоритм оператора `switch/case`

Оператори break, continue і goto

Оператор `break` використовується для виходу з оператора `while`, `do…while`, `for` і `switch`, що безпосередньо його містить. Управління передається на оператор, наступний за оператором, з якого здійснюється вихід. Приклад використання оператора `break` наведений вище.

Оператор `continue` використовується для ігнорування частини виконуваної ітерації циклу, який безпосередньо його містить, що залишилася. Якщо умовами циклу допускається нова ітерація, то вона виконується, інакше цикл завершується.

Оператор `goto` реалізує безумовний перехід, тобто дозволяє перейти в будь-яку точку програми, як вперед по тексту програми, так і назад. Точка переходу позначається за допомогою мітки, яка є довільним ідентифікатором з двокрапкою в кінці.

Оператори циклів

Оператори циклів використовують для виконання деякого фрагмента програми кілька разів. В окремих випадках фрагмент виконується в кожному послідовному кроці циклу без змін; частіше кожен крок циклу декілька відрізняється від попереднього.

Для грамотної реалізації будь-якого циклічного обчислювального процесу необхідно виконати дії, надані у вигляді узагальненого алгоритму на рис. 7.3.

Рис. 7.3. Узагальнений алгоритм циклічного процесу

Підготовка циклу полягає у визначенні початкових значень змін-

них, що будуть змінюватися в циклі, до початку виконання циклу.

Блок повторення – це дії, які повторюються в циклі. Вони завжди однакові, при цьому їх багаторазове повторення здійснюється при різних значеннях змінних циклу.

Модифікація – це зміна значень змінних циклу перед кожним новим повторення циклу.

Блок повторення та Блок модифікації разом складають *тіло циклу.*

Умова продовження циклу (команда переходу) полягає в перевірці умови продовження або закінчення циклу, тобто визначає скільки разів потрібно повторити тіло циклу.

Існує три види циклів: `while, for і do…while.`

Оператор циклу `while` називається циклом з передумовою та має такий формат:

```
while (вираз) тіло циклу;
```

Оператор `while` реалізує алгоритм, поданий на рис. 7.4.

Як вираз, допускається використовувати будь-який вираз мови С++, а як тіло – будь-який оператор, зокрема порожній або складений. Схема виконання оператора `while` така:

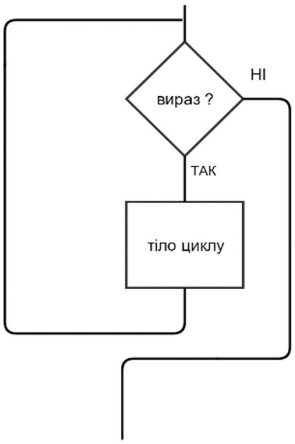


Рис. 7.4. Алгоритм оператора `while`

1. Обчислюється вираз.

2. Якщо вираз `false`, то виконання оператора `while` закінчується і виконується наступний за порядком оператор. Якщо вираз `true`, то виконується тіло циклу.

3. Процес повторюється з пункту 1.

4. Тіло циклу виконується доти, поки значення виразу дорівнює `true`.

5. Вираз обчислюється перед кожним виконанням оператора.

Цикл `for` має такий формат:

```
for (вираз 1; вираз 2; вираз 3;)
тіло циклу;
```

Оператор `for` реалізує алгоритм, поданий на рис. 7.5.

Вираз 1 зазвичай використовується для встановлення початкового значення змінних, які управляють циклом.

Вираз 2 – це вираз, що визначає умову, при якій тіло циклу виконуватиметься.

Вираз 3 визначає зміну змінних, що управляють циклом після кожного виконання тіла циклу.

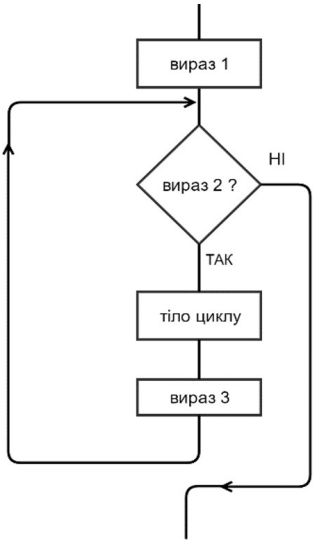


Рис. 7.5. Алгоритм оператора `for`

Схема виконання оператора `for`:

1. Обчислюється *вираз 1*.
2. Обчислюється *вираз 2*.
3. Якщо значення *виразу 2* відмінно від нуля (true), виконується тіло циклу.
4. Обчислюється *вираз 3* і здійснюється перехід до пункту 2.
5. Якщо *вираз 2* дорівнює нулю (`false`), то управління передається до оператора, наступного за оператором `for`.

Істотне те, що перевірка умови завжди виконується на початку циклу. Це означає, що тіло циклу може жодного разу не виконатися, якщо умова виконання відразу буде хибною.

Цикл `for` є зручним скороченим записом для циклу `while` вигляду:

```
expr 1;
while (expr 2) {
   body-of-loop;
   expr 3;
}
```

Вираз 1 задає початкові умови виконання циклу, *вираз 2* забезпечує перевірку умови виходу з циклу, а *вираз 3* модифікує умови, задані *виразом 1.*

Будь-який з виразів може бути опущений. Якщо опущено *вираз 2,* то за замовчуванням замість нього підставляється значення `true`.

Наприклад, цикл:

```
for (;вираз 2;) тіло циклу;
```

з опущеними *вираз 1* і *вираз 3* еквівалентний циклу:

```
while (вираз 2) тіло циклу;
```

Цикл:

```
for (;;) тіло циклу;
```

зі всіма опущеними виразами еквівалентний циклу:

```
while (true) тіло циклу;
```

тобто еквівалентний нескінченному циклу.

Такий цикл може бути перерваний тільки явним виходом з нього за допомогою операторів `break`, `goto` або `return`, що містяться в тілі циклу.

Оператор циклу `do while` називається оператором циклу з постумовою і використовується в тих випадках, коли необхідно виконати тіло циклу хоча б один раз. Формат оператора має такий формат:

```
do тіло циклу while (вираз);
```

Схема виконання оператора `do while`:

1. Виконується тіло циклу (яке може бути складеним оператором).

2. Обчислюється *вираз.*

3. Якщо *вираз* `false`, то виконання оператора `do while` закінчується і виконується наступний за порядком оператор. Якщо *вираз* `true`, то виконання оператора продовжується з пункту 1.

Щоб перервати виконання циклу до того, як умова стане хибною, можна використовувати оператор `break`.

Оператор `do while` реалізує алгоритм, наведений на рисунку 7.6.

На відміну від циклу `while`, у якому перевірка умови закінчення циклу робиться *до виконання* тіла циклу, у циклі `do while` така перевірка має місце *після виконання* тіла циклу. Отже, тіло циклу `do while` буде виконано *хоча б один раз*, навіть якщо вираз має значення `false` із самого початку.

Рис. 7.6. Алгоритм оператора `do while`

Лабораторний практикум № 7

Мета: набути практичних навичок з підготовки, налагодження та виконання програм на мові C++.

Завдання

Завдання 7.1

Надати математичний запис фрагмента програми та обчислити значення змінної x після його виконання, де N – це номер варіанта.

Таблиця 7.3.

Варіанти	Фрагмент

1–3	`x = 1;`
	`for (int j = 7; j>=N; j--)`
	`  x *= j;`
	`x *= 2;`

4–6	`x = 0;`
	`j = 1;`
	`do {`
	`  x += j;`
	`  j += 2;`
	`} while (j <=N);`

Продовж. табл. 7.3

Варіанти	Фрагмент

7–9	`x = 0;`
	`k = 3 * N;`
	`while (k > 0) {`
	`  x = sqrt(k+x);`
	`  k -= 3;`
	`}`

10–12	`x = N;`
	`for (int k=0; k<6; k++) {`
	`  if (k < 2) continue;`
	`  x++;`
	`}`

13–15	`x = 0;`
	`for (int j=0; j<N; j++)`
	`  x += 2;`
	`x *= 2;`

16–18	`x = 1;`
	`while (x <= N)`
	`  x++;`
	`x *= 2;`

19–21	`x = 1.0 / N;`
	`n = N;`
	`while (n > 1) {`
	`  n -= 2;`
	`  x = 1.0 / (n + x);`
	`}`

22–25	`x = 1.0 / N;`
	`n = N;`
	`k = 1;`
	`while (n > 1) {`
	`  n -= 2;`
	`  k = -k;`
	`  x = 1.0 / (n + k * x);`
	`}`

Завдання 7.2

Скласти програму табулювання функції f(x) на відрізку [a; b] з кроком h. Значення a, b, h вводити з клавіатури. Обов’язково врахувати область визначення функції.

Таблиця 7.4.

Варіант	Функція	Варіант	Функція
1	$y = \frac{tgx}{lnx}$	2	$y = \sqrt[3]{x}$
3	$y = tg\left(lnx\right)$	4	$y = \frac{ln\left(x-1\right)}{4-x}$
5	$y = tg^2\left(lnx\right)$	6	$y = ctg\left(lnx\right)$
7	$y = x^{0.2}$	8	$y = \frac{x}{1+tgx}$
9	$y = \frac{ln\left(x-0.5\right)}{\sqrt{x}}$	10	$y = \frac{sinx^3}{2-x}$
11	$y = \frac{cos^3 x}{1-lgx}$	12	$y = \frac{tgx}{lnx-1}$
13	$y = \frac{e^2}{\sqrt{1-x^2}}$	14	$y = \frac{x+1}{\sqrt{x}} - \sqrt[4]{\lvert x-2\rvert}$
15	$y = \frac{ln\lvert 2x\rvert}{sinx-\pi}$	16	$y = e^{lnx-1} + sinx$
17	$y = \frac{4-x}{ln\left(x-1\right)}$	18	$y = \frac{e^{x^2}}{\sqrt{1-x^2}}$

Завдання 7.3

Для заданих *x*, *n*, *eps*, що вводяться з клавіатури:

a. Обчислити *n* доданків згідно з варіантом.

b. Обчислити суму тих доданків, які за абсолютним значенням більше *eps*. (Завдання виконати для двох різних *eps*, які відрізняються на порядок, для кожного випадку обчислити кількість доданків).

c. Порівняти результати з «точним» значенням відповідної функції (сума визначає наближене значення) для x Є (-R,R).

Примітка 1. У цьому завданні значення x повинно належати області допустимих значень, визначеної в пункті c, значення n повинно бути досить великим (більше 20). eps потрібно обирати як маленьке додатне число, яке не більше 1e-9. Виконати обчислення для двох РІЗНИХ eps, що відрізняються не менше ніж на порядок (у 10 або більше разів).

Примітка 2. Результат виконання пункту b вважається задовільним, та програма може бути зарахована як правильна, лише за умови, що різниця між «точним» значенням функції та значеннями, обчисленими за *наближеними сумами, взята за модулем, є числом, що менше або дорівнює eps.*

Таблиця 7.5.

Варіант	Завдання
1	$\frac{sinx}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \ldots \left(R = \infty\right)$
2	$e^{-x^2} = 1 - \frac{x^2}{1!} + \frac{x^4}{2!} - \ldots + \left(-1\right)^n \frac{x^{2n}}{n!} \left(R = \infty\right)$
3	$ln\left(x + \sqrt{x^2+1}\right) = x - \frac{1}{2}\cdot\frac{x^3}{3} + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{x^5}{5} - \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\frac{x^7}{7} + \ldots \left(R = 1\right)$
4	$arctgx = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots \left(R = 1\right)$
5	$arcsinx = x + \frac{1}{2}\cdot\frac{x^3}{3} + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{x^5}{5} + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\frac{x^7}{7} + \ldots \left(R = 1\right)$
6	$\frac{1}{\sqrt{1-x^2}} = 1 + \frac{1}{2}\cdot x^2 + \frac{1}{2}\cdot\frac{3}{4}\cdot x^4 + \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot x^6 + \ldots \left(R = 1\right)$
7	$\frac{1}{\sqrt{1+x}} = 1 - \frac{1}{2}\cdot x + \frac{1}{2}\cdot\frac{3}{4}\cdot x^2 - \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot x^3 + \ldots \left(R = 1\right)$
8	$\sqrt{1+x} = 1 + \frac{1}{2}\cdot x - \frac{1}{2\cdot 4}\cdot x^2 + \frac{1\cdot 3}{2\cdot 4\cdot 6}\cdot x^3 - \ldots \left(R = 1\right)$

Продовж. табл. 7.5

Варіант	Завдання
9	$\frac{1}{(1+x)^3} = 1 - \frac{2 \cdot 3}{2} \cdot x + \frac{3 \cdot 4}{2} \cdot x^2 - \frac{4 \cdot 5}{2} \cdot x^3 + \ldots (R = 1)$
10	$\frac{1}{(1+x)^2} = 1 - 2x + 3x^2 - 4x^3 + 5x^4 - \ldots (R = 1)$
11	$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - \ldots (R = 1)$
12	$ln\frac{1+x}{1-x} = 2 \cdot \left(x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} + \ldots \right) (R = 1)$
13	$ln(1-x) = -\frac{x}{1} - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \ldots (R = 1)$
14	$ln(1+x) = \frac{x}{1} - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} - \ldots (R = 1)$
15	$ch(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \ldots (R = \infty)$
16	$sh(x) = 1 + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots (R = \infty)$
17	$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots (R = \infty)$
18	$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots (R = \infty)$
19	$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots (R = \infty)$
20	$e^{-x} = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots (R = \infty)$

Контрольні питання

1. Опишіть структуру програми мовою C++.
2. Наведіть приклади ключових слів мови C++.
3. Які прості типи даних існують у мові C++?
4. Опишіть правила приорітету операцій у C++.
5. Які оператори розгалуження є в мові C++?
6. Який оператор розгалуження є в C++, такий, якого немає в Kotlin?
7. Які оператори циклів є в мові C++?
8. У чому основна відмінність циклів for у мовах C++ та Kotlin?
9. Яка особливість циклу do..while дозволена в Kotlin та не дозволена в C++?
10. Для чого в мові C++ є оператор goto? Чому його рекомендовано уникати?

Тема 8. МАСИВИ. ФУНКЦІЇ. РЕКУРСІЯ

Масиви в С++

Масив – це сукупність логічно взаємозв'язаних даних одного типу, об'єднаних загальним ім'ям. Масив у С++ – це набір однотипних даних (об'єктів), що мають загальне ім'я і що розрізняються місцеположенням у цьому наборі (або індексом, який присвоюється кожному елементу масиву). У програмі можуть бути масиви цілих чисел або чисел з плаваючою точкою, символьні масиви і так далі.

Опис одновимірного масиву

Опис складається із специфікації типу, ідентифікатора й розмірності масиву. Звернення до елементів масиву виконується за їх індексами, які завжди починаються від нуля. Якщо, наприклад, оголосити в програмі масив на ім'я `mas` з 100 цілих чисел `int mas[100];` то найперший елемент масиву отримує позначення `mas[0]`, наступний – `mas[1]` і так далі до останнього елемента `mas[99]`. Так само можна оголосити масиви даних будь-яких інших типів:

```
char c[20];           // Масив з 20 символів
float f[16];          // Масив з 16 чисел з плаваючою точкою.
//Після оголошення масиву його можна заповнити числами,
використовуючи індексний оператор [ ], наприклад:
int myArr[3];          // Оголошення масиву
myArr[0] = -12;        // Ініціалізація першого елемента
// (його індекс дорівнює нулю) значенням –12
myArr[1] = 5;
myArr[2] = 37;
```

Доступ до окремих елементів масиву здійснюється також за допомогою індексного оператора:

```
int result = myArr[0]  + myArr[1] + myArr[2];
```

Масив можна оголосити і заповнити одночасно, наприклад:

```
int myArr[3]  =  { -12, 5, 37 } ;
```

Якщо розмірність задана, то число значень у списку ініціалізації не повинне її перевищувати. Якщо розмірність масиву більше числа значень у списку, то елементи масиву, що не ініціалізували явно, будуть установлені в 0, наприклад:

```
int myArr[5]={ 0, 1, 2 }; //myArr буде дорівнювати
{0, 1, 2, 0, 0}
```

За давнім стандартом, значення розмірності має бути константним виразом, тобто розмірність має бути відома на етапі трансляції. Це означає, що змінна не може використовуватися для завдання розмірності масиву. Проте сучасні версії компіляторів можуть дозволяти використовувати змінні як розмір масиву. Звичайно, при цьому не гарантується робота програми із використанням усіх компіляторів. Якщо в масиві не дуже багато елементів і їх значення відомі заздалегідь, масив можна ініціалізувати разом з його оголошенням, уклавши перелік значень у фігурні дужки:

```
int ns[10]={0,1,2,3,4,5,6,7,8,9};
```

Якщо ж масив великий, заповнити його доведеться програмно в циклі. Нехай ми хочемо утворити масив зі 100 елементів, заповнених натуральним рядом чисел. Це робиться таким чином:

```
int mas[100];
for (int i=0;i<100;i++) mas[i]=i+1;
```

До елементів масиву можна звертатися і вибірково, наприклад:

```
test[50]=0x7FFA;
```

Збереження одновимірних масивів

У C++ одновимірний масив логічно зберігається як послідовність впорядкованих елементів у пам'яті (рис. 8.1).

Кожен елемент відноситься до одного і того ж типу даних.

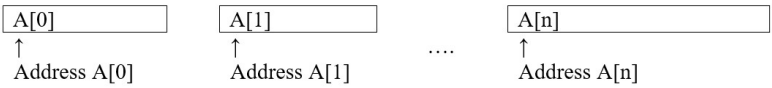


Рис. 8.1. Представлення одновимірного масива в пам'яті

У C++ ім'я масиву є константою і розглядається як адреса першого елемента цього масиву. Так, в оголошенні `type A[arraySize];` ім'я масиву A є константою і визначає місцеположення в пам'яті першого елемента `A[0]`. Елементи `A[1], A[2]` і т. д. йдуть за ним послідовно. Припустимо, що `sizeof(type) = M`, тоді весь масив `A` займає `M * arraySize` байт. Компілятор задає таблицю, названу дескриптором масиву, для ведення запису характеристик масиву.

Символьні масиви

Символьні масиви можуть становити набір окремих символів або зв'язний символьний рядок. У першому випадку масив можна ініціалізувати так само, як і числовий, перерахуванням усіх його елементів:

```
char     cs[17]={'С','и','м','в','о','л','ь','н','и',
'й',' ','м', 'а','с','и','в'};
```

Такий масив займає в пам'яті точно 16 байт. У другому випадку масив ініціалізувався символьним рядком:

```
char cz[20] = "Символьний масив";
```

За такої ініціалізації компілятор записує в пам'ять у кінці рядка двійковий нуль, який є символом-обмежувачем. Багато функцій для роботи з рядками (наприклад,

копіювання рядка або виведення рядка на екран) за цим символом визначають, де закінчується рядок. Це позбавляє нас від необхідності визначати і вказувати в разі виклику функції фактичну довжину рядка. Таким чином, масив займає в пам'яті на 1 байт більше, ніж у ньому є значущих символів. Цю обставину необхідно враховувати під час завдання довжини масиву в його оголошенні, яка повинна вибиратися на одиницю більше максимально можливої довжини рядка. Оголошення довжини символьного масиву із запасом доцільно лише в тих випадках, коли він заповнюватиметься програмно рядками різної довжини. Якщо ж даний текстовий рядок змінюватися не буде, зручніше оголосити її без рекомендації довжини, залишивши в оголошенні пару квадратних дужок, які скажуть компілятору, що змінна cz є не одиничним символом, а символьним масивом:

```
char cz[] = "Символьний масив";
```

Компілятор, виділяючи пам'ять під цей масив, сам визначить його довжину, яка в даному випадку дорівнюватиме 17 байтам.

Приклади операцій з лінійним масивом показані в наступній програмі:

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

#define N 10

using namespace std;
int main ()
{
    int m[N];

    // 1. Ініціалізація масиву нулями.
    for (int i = 0; i < N; i ++) m [i] = 0;
```

```
    // 2. Ініціалізація масиву випадковими числами.
    for (int i = 0; i < N; i ++) m [i] = rand()%11;

    // 3. Друк масиву в рядок.
   for (int i = 0; i < N; i ++) cout << setw(4) << m[i];
    cout << endl;

    // 4. Обчислення суми елементів масиву.
    int sum = 0;
    for (int i = 0; i < N; i++) sum += m[i];
    cout << "Сума = " << sum << endl;

     // 5. Виведення елементів масиву у вигляді гісто-
грами.
      cout << "Елемент" << setw(13) << "Значення" <<
setw(13) << "Гістограма" << endl;
    for (int i = 0; i < N; i ++) {
         cout << setw(7) << i << setw(13) << m[i] <<
"..." ;
        for (int j = 0; j < m[i]; j++ ) cout << "*";
        cout << endl;
    }

    // 6. Сортування масиву методом "бульбашки".
    int temp;
    cout << "Початковий порядок елементів" << endl;
    for (int i = 0; i < N; i ++) cout << setw(4) << m[i];
    cout << endl;
    // сортування
    for (int i = 0; i<N-1; i++ )
        for (int j = i; j<N; j++ )
            if (m[i]> m[j]) {
                temp = m[i]; m[i]= m[j]; m[j]= temp;
            }
    cout << "Масив після сортування" << endl;
    for (int i = 0; i < N; i ++) cout << setw(4) << m[i];
    cout << endl;
```

```
    // 7. Лінійний пошук в масиві.
    int key, flag = N;
    cout << "Введіть ключ пошуку" << endl;
    cin >> key;
    for (int i = 0; i < N; i ++) {
        if (m[i] == key ) {
            flag = i;
            cout << "Шуканий елемент має індекс=" <<
i << endl;
        }
    }
    if (flag == N )
        cout << "Елемент не знайдений" << endl;

    // 8. Пошук максимального і мінімального елементів
масиву.
    int max, min;
    max = min = m[0];
    for (int i = 1; i < N; i++) {
        if (m[i] > max) max = m[i];
        if (m[i] < min) min = m[i];
    }
    cout << "Максимальний елемент=" << max << endl;
    cout << "Мінімальний елемент =" << min << endl;


     // 9. Перестановка елементів масиву у зворотному
порядку.
    for (int i = 0; i<(N/2); i++ ) {
        temp = m[i];
        m[i]=m[N-1-i];
        m[N-1-i]=temp;
    }
    // друк
    cout<< "Перестановка\n";
    for (int i = 0; i < N; i ++) cout << setw(4) << m[i];
    cout << endl;
    return 0;
}
```

Якщо запустити цю програму на виконання, результат може бути таким:

```
10  10   7   9   7   3   1   9   8   2
Сума = 66
Елемент     Значення   Гістограма
      0           10...**********
      1           10...**********
      2            7...*******
      3            9...*********
      4            7...*******
      5            3...***
      6            1...*
      7            9...*********
      8            8...********
      9            2...**
Початковий порядок елементів
  10  10   7   9   7   3   1   9   8   2
Масив після сортування
   1   2   3   7   7   8   9   9  10  10
Введіть ключ пошуку
8
Шуканий елемент має індекс=5
Максимальний елемент=10
Мінімальний елемент =1
Перестановка
  10  10   9   9   8   7   7   3   2   1

Process finished with exit code 0
```

Функції в C++

Функція – це іменована послідовність описів і операторів, що виконує яку-небудь закінчену дію. Функція може приймати параметри й повертати значення.

Будь-яка програма на C++ складається з функцій, одна з яких повинна мати ім'я `main` (з неї починається виконання програми). Функція починає виконуватися в момент виклику. Будь-яка функція має бути оголошена й визначена.

Оголошення функції повинно знаходитися в тексті раніше її виклику для того, щоб компілятор міг здійснити перевірку правильності виклику.

Оголошення функції (прототип, заголовок, сигнатура) задає її ім'я, тип значення, що функція повертає і список параметрів.

Визначення функції містить, окрім оголошення, тіло функції, що є послідовністю операторів і описів у фігурних дужках:

```
[ клас ]  тип ім'я  ( [ список_параметрів ] ) {
      // тіло функції
}
```

Тип значення, яке повертає функція, може бути будь-яким, окрім масиву й функції (але може бути покажчиком на масив або функцію). Якщо функція не повинна повертати значення, указується тип `void`.

Список параметрів визначає величини, які потрібно передати у функцію під час її виклику. Елементи списку параметрів розділяються комами. Для кожного параметра, що передається у функцію, указується його тип та ім'я (у оголошенні імена можна опускати).

У визначенні, в оголошенні і під час виклику однієї і тієї ж функції типи й порядок слідування параметрів повинні збігатися.

На імена параметрів обмежень по відповідності не накладається, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони використовуються тільки для поліпшення читаності програми).

Функцію можна визначити як вбудовану за допомогою модифікатора `inline`, який рекомендує компілятору замість звернення до функції поміщати її код безпосередньо в кожну точку виклику. Модифікатор `inline` ставиться перед ти-

пом функції. Він застосовується для коротких функцій, щоб знизити накладні витрати на виклик (збереження і відновлення регістрів, передача управління). Директива `inline` носить рекомендаційний характер і виконується компілятором за потреби. Використання `inline` – функцій може збільшити об'єм виконуваної програми. Визначення функції повинно передувати її викликам, інакше замість `inline` – розширення компілятор згенерує звичайний виклик.

Тип значення, що повертає функція і типи параметрів спільно визначають тип функції.

Для виклику функції в простому випадку потрібно вказати її ім'я, за яким у круглих дужках через кому перераховуються імена аргументів, що передаються. Виклик функції може знаходиться в будь-якому місці програми, де по синтаксису допустимий вираз того типу, який формує функція. Якщо тип значення, яке повертає функція не `void`, то вона може входити до складу виразу або, в окремому випадку, розташовуватися в правій частині оператора присвоювання.

Розглянемо приклад програми, що знаходить найменше з трьох чисел, використовуючи при цьому функцію `min3()`, викликаючи її за іменем.

```
#include <iostream>

int min3(int x, int y, int z);

using namespace std;

int main() {
    int a,b,c;
    cin >> a >> b >> c;
    int x = min3(a,b,c);
    cout << "min = " << x << endl;
    return 0;
}
```

```
int min3(int x, int y, int z) {
    return x < y ? x < z ? x : z : y < z ? y : z;
}
```

Під час виконання програми можна одержати такий результат:

```
4 7 3
min = 3

Process finished with exit code 0
```

Особливості виконання функцій

Усі величини, які описані всередині функції, а також її параметри, є локальними. Зоною їх дії є функція. Під час виклику функції, як і при вході в будь-який блок, у стеку виділяється пам'ять під локальні автоматичні змінні. Крім того, у стеку зберігається вміст регістрів процесора на момент, що передує виклику функції, і адресу повернення з функції для того, щоб після виходу з неї можна було продовжити виконання функції, яка її викликала.

При виході з функції відповідна ділянка стека звільняється, тому значення локальних змінних між викликами однієї і тієї ж функції не зберігаються. Якщо цього потрібно уникнути, під час оголошення локальних змінних використовується модифікатор `static` (проте, це робити не рекомендується).

Значення, що повертається функцією

Механізм повернення з функції у функцію, що викликала її, реалізується оператором:

```
return [ вираз ];
```

Якщо функція описана як `void`, вираз не вказується. Оператор `return` використовується для дострокового виходу з функції.

Оператор `return` можна опускати для функції типу `void`, якщо повернення з неї відбувається перед фігурною дужкою, що закривається, і для функції `main`.

Вираз, указаний після `return`, неявно перетвориться до типу значення, що повертається функцією, і передається в точку виклику функції.

Функція може містити декілька операторів `return` (це визначається потребами алгоритму).

Обмін інформацією між функціями

Під час спільної роботи функції повинні обмінюватися інформацією. Це можна здійснити:

- за допомогою глобальних змінних;
- через параметри функції;
- через значення, що повертає функція.

Обмін інформацією за допомогою глобальних змінних

Глобальні змінні видно у всіх функціях, де не описані локальні змінні з тими ж іменами, тому використовувати їх для передачі даних між функціями дуже легко. Проте, це не рекомендується, оскільки ускладнює відлагодження програми й перешкоджає розміщенню функцій у бібліотеки загального користування. Потрібно прагнути до того, щоб функції були максимально незалежні, а їх інтерфейс повністю визначався прототипом функції.

Використання параметрів функції для обміну інформацією між функціями

Механізм параметрів є основним способом обміну інформацією між функціями, що викликаються і викликають. Параметри, перераховані в заголовку опису функції, називаються *формальними*, а записані в операторі виклику функції – *фактичними*.

Під час виклику функції насамперед обчислюються вирази, що стоять на місці фактичних параметрів; потім у стеку виділяється пам’ять під формальні параметри функції відповідно до їх типу, і кожному з них привласнюється значення відповідного фактичного параметра. До того ж

перевіряється відповідність типів і за потреби виконуються їх перетворення. У разі невідповідності типів видається діагностичне повідомлення.

Існує два способи передачі параметрів у функцію: за значенням і за адресою.

Під час передачі за значенням у стек заносяться копії значень фактичних параметрів, і оператори функції працюють з цими копіями. Доступу до початкових значень параметрів у функції немає, а, отже, немає і можливості їх змінити. Під час передачі за адресою у стек заносяться копії адрес параметрів, а функція здійснює доступ до елементів пам'яті по цих адресах і може змінити початкові значення параметрів.

```
// Способи передачі параметрів у функцію
#include <iostream>
using namespace std;

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 << endl;
    f(i, &j, k);
    cout << i << " " <<  j << " " <<  k <<endl;
    return 0;
}

void f (int i, int* j, int& k)
{
    i++; (*j)++; k++;
}
```

Результат роботи програми:

```
i j k
1 2 3
1 3 4

Process finished with exit code 0
```

Перший параметр (i) передається за значенням. Його зміна у функції не впливає на початкове значення. Другий параметр (j) передається за адресою за допомогою покажчика, до того ж для передачі у функцію адреси фактичного параметра використовується операція взяття адреси, а для набуття його значення у функції потрібна операція розіменування. Третій параметр (k) передається за адресою за допомогою посилання.

У разі передачі за посиланням, у функцію передається адреса вказаного під час виклику параметра, а всередині функції всі звернення до параметра неявно «розіменуються». Тому використання посилань замість покажчиків покращує читабельність програми, позбавляючи від необхідності застосовувати операції отримання адреси й розіменування.

Використання посилань замість передачі за значенням ефективніше, оскільки не вимагає копіювання параметрів, що має значення під час передачі структур даних великого об'єму.

Якщо потрібно заборонити зміну параметра всередині функції, використовується модифікатор `const`, наприклад:

```
int f(const char*);
```

Рекомендується вказувати `const` перед усіма параметрами, зміна яких у функції не передбачається. Це полегшує відлагодження великих програм, оскільки за заголовком функції можна зробити висновок про те, які величини в ній змінюються, а які ні. Крім того, на місце параметра типу `const&` може передаватися константа, а для змінної за потреби виконуються перетворення типу.

Таким чином, початкові дані, які не повинні змінюватися у функції, бажано передавати їй за допомогою константних посилань. За замовчуванням параметри будь-якого типу,

окрім масиву й функції (наприклад, дійсного, структурного, перерахування, об'єднання, покажчик), передаються у функцію за значенням.

Перевантаження функцій

У мові C++ передбачена можливість створення декількох функцій з однаковим іменем. Це називається *перевантаженням функцій*. Перевантажені функції повинні відрізнятися одна від одної списками параметрів: або типом одного або декількох параметрів, або різною кількістю параметрів, або і тим і іншим одночасно. Розглянемо такий приклад:

```
int  func(int, int);
int  func(long, long);
int  func(long);
```

Функція `func()` перевантажена з трьома різними списками параметрів. Перша і друга версії відрізняються типами параметрів, а третя – їх кількістю.

Типи значень, що повертають перевантажені функції, можуть бути однаковими або різними. Слід мати на увазі, що створення двох функцій з однаковим іменем і однаковим списком параметрів, але з різними типами значень, що повертаються, не дозволяється та призведе до помилки компіляції.

Припустимо, потрібно створити функцію, яка подвоює будь-яке отримане нею значення. До того ж хотілося б мати можливість передавати їй значення типу `int`, `long`, `float` або `double`. Без перевантаження функцій довелося б створювати чотири різні функції:

```
int Doubleint(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

За допомогою перевантаження функцій можна використовувати такі оголошення:

```
int Double(int);
long Double(long);
float DouDle(float);
double Double(double);
```

Завдяки використанню перевантажених функцій не потрібно турбуватися про виклик у програмі потрібної функції, що відповідає типу змінних, які передаються. Під час виклику перевантаженої функції компілятор автоматично визначить, який саме варіант функції слід використовувати.

Параметри функції, використовувані за замовчуванням

Для кожного параметра, що оголошується в прототипі і визначенні функції, має бути передане відповідне значення у виклику функції.

Передаване значення повинно мати оголошений тип. Отже, якщо деяка функція оголошена як `long func(int);` то вона дійсно повинна набувати цілочисельного значення. Якщо тип оголошеного параметра не збіжиться з типом передаваного аргументу, компілятор повідомить про помилку.

З цього правила існує одне виключення, яке набуває чинності, якщо в прототипі функції для параметра оголошується стандартне значення. Це значення, яке використовується в тому випадку, якщо під час виклику функції для цього параметра не встановлено ніякого значення. Дещо змінимо попереднє оголошення:

```
long func(int  x = 50);
```

Прототип такого оголошення потрібно розуміти таким чином: функція `func(int)` повертає значення типу `long` і приймає параметр типу `int`. Але якщо під час виклику цієї функції аргумент наданий не буде, використовуйте замість нього число 50. А оскільки в прототипах функцій

імена параметрів не обов'язкові, то останній варіант оголошення можна переписати по-іншому:

```
long func(int = 50);
```

Визначення функції не змінюється під час оголошення значення параметра, що задається за замовчуванням. Тому заголовок визначення цієї функції виглядатиме так, як і раніше:

```
long func(int  x);
```

Якщо під час виклику цієї функції аргумент не встановлюється, то компілятор привласнить змінною x значення 50. Ім'я параметра, для якого в прототипі встановлюється значення за замовчуванням, може не збігатися з ім'ям параметра, що вказується в заголовку функції: значення, задане за замовчуванням, присвоюється за позицією, а не за іменем.

Установку значень за замовчуванням можна призначити будь-яким або всім параметрам функції. Але одне *обмеження* все ж таки діє: *якщо якийсь параметр не має стандартного значення, то жоден з попередніх щодо нього параметрів також не може мати стандартного значення*.

Рекурсія

Загалом рекурсія значить самоповторюваний шаблон. У математиці це може бути функція визначена через себе. Інакше кажучи, це функція, що викликає сама себе. Кожна рекурсивна функція має умову завершення; інакше вона буде викликати себе безперестанку. Таку умову називають *базовою умовою*.

Типи рекурсії. Алгоритм рекурсії може бути реалізовним більш ніж одним способом. Можливі варіанти рекурсії – це *лінійна*, *хвостова*, *обопільна*, *двійкова* і *вкладена*. Ви можете здійснити їх або під час компіляції через використання шаблонів або під час виконання через використання функцій.

Лінійна рекурсія

Лінійна рекурсія – це найпростіший вид рекурсії і, можливо, найуживаніша. У цієї рекурсії одна функція просто викликає себе доти, доки не досягне умови завершення (також відомої як базова умова); цей процес відомий як намотування. Як тільки виконана умова завершення, виконання програми повертається до викликальника; це зветься *розмотуванням*.

Упродовж намотування і розмотування функція може виконувати якісь додаткові корисні задачі; у випадку факторіала вона множить вхідне значення на значення повернуте під час фази розмотування. Цей процес можна зобразити у вигляді такої діаграми, яка показує обидві фази функції обчислення факторіала із використанням лінійної рекурсії (рис. 8.2).

Математично ви можете написати функцію обчислення факторіала таким чином; інакше кажучи, коли значення «n» нуль, повертати одиницю і, коли значення «n» більше ніж нуль, викликати функцію рекурсивно з «n-1» і помножити результат на «n».

$$f(n) = \begin{cases} 1, & n = 0 \\ n * f(n-1), & n > 0 \end{cases}$$

```
int Factorial(int n)
{
  // умова завершення
  if (n == 0)
  return 1;

  // лінійний рекурсивний виклик
  return n * Factorial(n - 1);
}
```

Програма становить собою реалізацію лінійної рекурсії часу виконання. Тут ми маємо умову завершення

у вигляді 0; програма починає виконувати розмотування, коли досягає умови завершення.

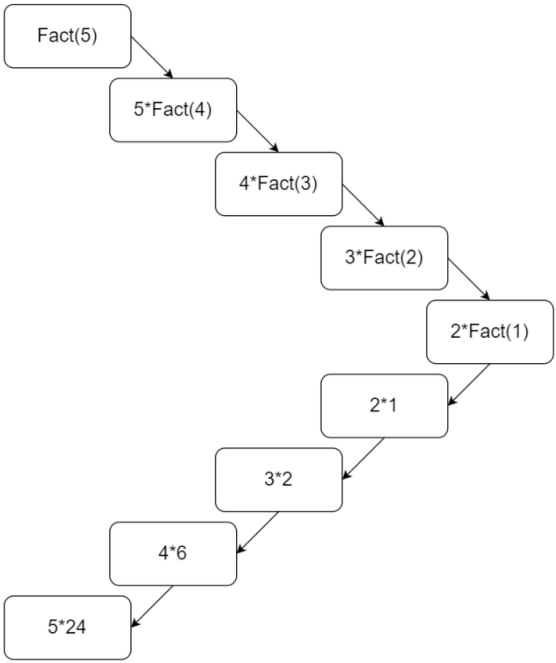


Рис. 8.2. Обчислення факторіала
з використанням лінійної рекурсії

Хвостова рекурсія

Хвостова рекурсія – це спеціальна форма лінійної рекурсії, де рекурсивний виклик зазвичай іде останнім у функції. Цей тип рекурсії здебільшого більш ефективний, бо розумні компілятори автоматично перетворять таку рекурсію в цикл задля уникнення вкладених викликів функцій. Через те, що рекурсивний виклик функції зазвичай останнє, що робить функція, вона не має потреби ще щось робити під час розмотування; натомість, вона просто повертає

значення отримане через рекурсивний виклик. Ось приклад тієї самої програми, реалізованої як хвостова рекурсія (рис. 8.3).

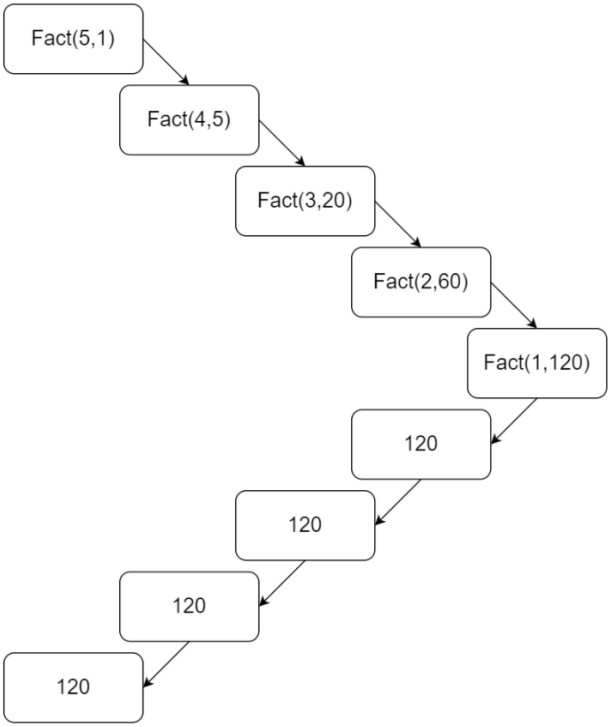


Рис. 8.3. Обчислення факторіала
з використанням хвостової рекурсії

Ви можете визначити хвостову рекурсію математично через наступну формулу; інакше кажучи, коли значення «n» нуль, просто повернути значення «a»; якщо значення «n» більше ніж нуль, викликати рекурсивну функцію з параметрами «n-1» і «n*a». Також можна зауважити, що під час фази розмотування кожна рекурсивно викликана функція просто повертає значення «a».

$$f(n,a)=\begin{cases} a, & n=0 \\ f(n-1,n*a), & n>0 \end{cases}$$

```
int Factorial(int n, int a)
{
   // умова завершення
   if (n==0)
      return a;
   // хвостовий рекурсивний виклик
   return Factorial(n - 1, n * a);
}
```

Це змінена версія програми з лінійною рекурсією. Ви виконуєте всі обчислення до виклику рекурсивної функції, і просто повертаєте значення отримане з цього виклику. Тут порядок обчислення зворотній до порядку за лінійної рекурсії. У випадку лінійної рекурсії, ви спочатку множите 1 на 2; отриманий результат на 3 і так далі. З іншого боку, тут ви множите n на n-1, і тоді на n-2, доки не досягнете 0.

Хвостова рекурсія дуже корисна і часом неуникна у функціональних мовах програмування, бо деякі з них можуть не підтримувати циклічні конструкції. Тоді зазвичай цикли реалізуються за допомогою хвостової рекурсії. За допомогою хвостової рекурсії ви можете робити майже все, що можна зробити з циклом, але у зворотному напрямку це часто неправильно. Ось дуже простий приклад, що демонструє цикл через хвостову рекурсію.

```
// реалізація циклу через хвостову рекурсію
// проста версія
void RecursiveLoop(int n)
{
   // умова завершення
   if (n == 0)
      return;

   // дія
   cout << n << endl;
```

```
    // хвостовий рекурсивний виклик
    return RecursiveLoop(--n);
}
```

Обопільна рекурсія

Обопільна рекурсія також відома як непряма рекурсія. У цьому типі рекурсії, дві або більше функції викликають одна одну циклічно. Це єдиний шлях для здійснення рекурсії в мовах, що не дозволяють вам викликати функції рекурсивно. Умова завершення в такій рекурсії може бути в одній або всіх функціях (рис. 8.4).

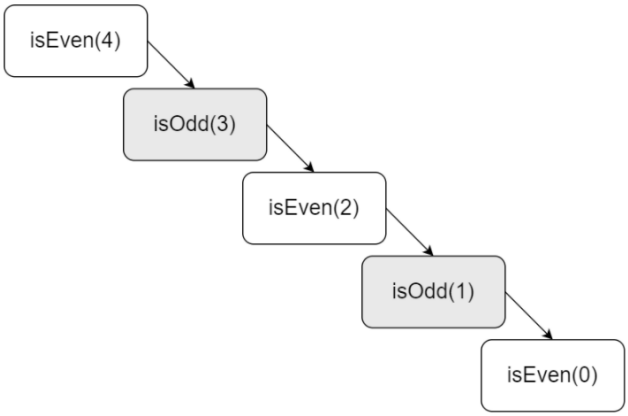


Рис. 8.4. Обчислення з використанням обопільної рекурсії

Математично ви можете визначити ці функції як:

$$f(n)=\begin{cases} true, & n=0 \\ isOdd(n-1), & n>0 \end{cases}$$

$$f(n)=\begin{cases} false, & n=0 \\ isEven(n-1), & n>0 \end{cases}$$

```
bool isEven(int no)
{
   // умова завершення
   if (n == 0)
      return true;
   else
      // взаємний рекурсивний виклик
      return isOdd(n - 1);
}

bool isOdd(int n)
{
   // умова завершення
   if (n == 0)
      return false;
   else
      // взаємний рекурсивний виклик
      return isEven(n - 1);
}
```

Визначення парності числа за допомогою обопільної рекурсії не дуже гарна ідея. Більш цікавим прикладом є чоловіча й жіноча послідовності. Обидві функції рекурсивно викликають одна одну й можуть бути представлені так.

$$Male(n) = \begin{cases} 0, & n = 0 \\ n - Female\big(Male(n-1)\big), & n > 0 \end{cases}$$

$$Female(n) = \begin{cases} 1, & n = 0 \\ n - Male\big(Female(n-1)\big), & n > 0 \end{cases}$$

```
int MaleSequence(int n)
{
   // умова завершення
   if (n == 0)
      return 0;
```

```
    // взаємний рекурсивний виклик
    return n - FemaleSequence(MaleSequence(n-1));
}

int FemaleSequence(int n)
{
    // умова завершення
    if (n == 0)
        return 1;

    // взаємний рекурсивний виклик
    return n - MaleSequence(FemaleSequence(n-1));
}
```

Двійкова рекурсія

У випадку двійкової рекурсії функція викликає себе двічі, замість одного разу. Такий тип рекурсії дуже корисний під час роботи з деякими структурами даних, наприклад, під час обходу дерева в прямому, зворотному або центрованому порядку, або генерації чисел Фібоначчі (рис. 8.5) і так далі.

Двійкова рекурсія – це особлива форма експонентної рекурсії, де одна функція викликає себе більш ніж один раз (у випадку двійкової рекурсій).

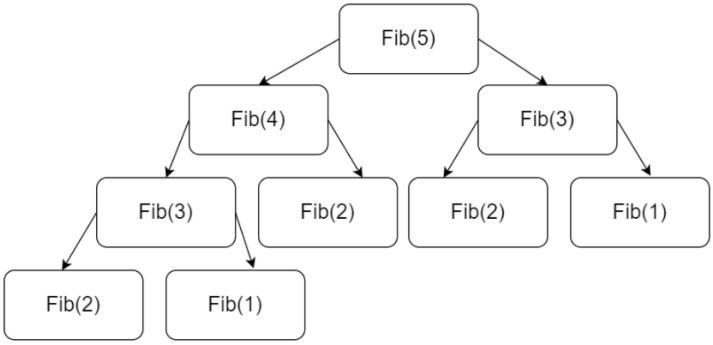


Рис. 8.5. Двійкова рекурсія
на прикладі визначення послідовності Фібоначчі

Математично ви можете визначити послідовність Фібоначчі як:

$$f(n)=\begin{cases}0, & n=0\\ 1, & n=1\\ f(n-1)+f(n-2), & n>1\end{cases}$$

```
int Fib(int n)
{
   // умова завершення
   if (n==0 || n==1)
      return n;

   // подвійний рекурсивний виклик
   return Fib(n - 1) + Fib(n - 2);
}
```

Ось проста реалізація послідовності Фібоначчі, що викликає рекурсивну функцію двічі. Тут ми маємо два базові випадки; коли значення параметра на вході є 0 чи 1. Це, звісно, не найкраща реалізація послідовності Фібоначчі, і ви можете перетворити її у хвостову рекурсію, трошки змінивши її.

```
int Fib(int n, int a = 0, int b = 1)
{
// умова завершення
   if (n == 1)
      return b;
   else
      // хвостовий рекурсивний виклик
      return Fib(n-1, b, a+b);
}
```

Тут ви перетворюєте двійкову рекурсію у хвостову. Ви просто робите обчислення перед рекурсивним викликом; звідси, ви не маєте двічі робити рекурсивний виклик.

Математично це можна виразити як:

$$f(n,a,b)=\begin{cases} n, & при\ n\in[0,1] \\ f(n-1,b,a+b), & при\ n>1 \end{cases}$$

Вкладена рекурсія

Це особливий тип рекурсії, коли рекурсивні виклики вкладені. В усіх попередніх типах рекурсії, ви можете замінити рекурсію на простий цикл або цикл зі стеком, але цей тип рекурсії не може бути легко замінений на простий цикл.

Типовим прикладом вкладеної рекурсії є функція Акермана.

Математично функція Акермана може бути визначена як:

$$A(m,n)=\begin{cases} n+1, & m=0 \\ A(m-1,1), & (m>0)\,and\,(n=0) \\ A(m-1,A(m,n-1)), & (m>0)\,and\,(n>0) \end{cases}$$

```
int Ackermann(int m, int n)
{
   // умова завершення
   if (m == 0)
      return n + 1;

   // лінійний рекурсивний виклик
   else if (m > 0 && n == 0)
      return Ackermann(m-1, 1);

   // вкладений рекурсивний виклик
   else
      return Ackermann(m-1, Ackermann(m, n-1));
}
```

Ця функція має дві умови завершення; одна умова припиняє вкладені виклики й починає лінійну рекурсію; друга умова завершення припиняє лінійну рекурсію.

Лабораторний практикум № 8

Мета: набути практичних навичок розробки програм, що використовують списки та/або масиви.

Завдання

Завдання 8.1

Розв'язати задачу за посиланням у табл. 8.1. До звіту включити код програми, результат виконання контрольного прикладу (зі сторінки задачі) та посилання на сторінку результату успішного проходження вашою програмою всіх тестів.

Наприклад, для задачі: https://www.eolymp.com/uk/problems/8975, така сторінка може знаходитись за посиланням: https://www.eolymp.com/uk/submissions/6963953.

Таблиця 8.1.

Варіант	Завдання
1	https://www.eolymp.com/uk/problems/7843
2	https://www.eolymp.com/uk/problems/7844
3	https://www.eolymp.com/uk/problems/914
4	https://www.eolymp.com/uk/problems/917
5	https://www.eolymp.com/uk/problems/928
6	https://www.eolymp.com/uk/problems/7831
7	https://www.eolymp.com/uk/problems/7832
8	https://www.eolymp.com/uk/problems/8959
9	https://www.eolymp.com/uk/problems/8961
10	https://www.eolymp.com/uk/problems/8962
11	https://www.eolymp.com/uk/problems/7833
12	https://www.eolymp.com/uk/problems/7848
13	https://www.eolymp.com/uk/problems/8680
14	https://www.eolymp.com/uk/problems/7845
15	https://www.eolymp.com/uk/problems/5059

Продовж. табл. 8.1

Варіант	Завдання
16	https://www.eolymp.com/uk/problems/7845
17	https://www.eolymp.com/uk/problems/2238
18	https://www.eolymp.com/uk/problems/7537
19	https://www.eolymp.com/uk/problems/1952
20	https://www.eolymp.com/uk/problems/7834

Завдання 8.2

Програма повинна вводити дані, викликати функцію, що виконує дію, описану в табл. 8.2 та виводить результат (якщо іншого не вказано в явному вигляді). Для виконання дії, указаної в умові, використовувати рекурсивну функцію. Використання циклів та рядкових функцій ЗАБОРОНЕНО.

Таблиця 8.2.

Варіант	Завдання
1	Дано натуральне число. Вивести його цифри у зворотному порядку, розділяючи їх пробілами
2	Дано натуральне число. Вивести його цифри в прямому порядку, розділяючи їх пробілами
3	Дано натуральне число n. Вивести натуральні числа від 1 до n
4	Дано натуральне число n. Вивести натуральні числа від n до 1
5	Дано два цілих числа a та b. Вивести всі цілі числа від a до b включно, у порядку зростання
6	Дано два цілих числа a та b. Вивести всі цілі числа від a до b включно, у порядку спадання
7	Дано натуральне число n. Визначте кількість його цифр
8	Дано натуральне число n. Визначте суму його цифр
9	Дано натуральне число n. Визначте кількість його непарних цифр
10	Дано натуральне число n. Визначте суму його парних цифр

Продовж. табл. 8.2

Варіант	Завдання
11	Дано натуральне число n. Визначте суму його непарних цифр
12	Дано натуральне число n. Визначте кількість його парних цифр

Завдання 8.3

Скласти програму обчислення наступних величин (табл. 8.3) та виконати її в середовищі програмування. Для розв'язання основної задачі (крім формування початкового масиву та введення-виведення) використовувати цикли ЗАБОРОНЕНО.

Елементи масива визначаються за формулою

```
a[i] = p[i] – 64;
```

де `p[i+1] = (p[i] * 67 + 11) % 128`.

p[0] дорівнює N – номеру варіанта за списком групи, кількість елементів у масиві дорівнює 50.

Таблиця 8.3.

Варіант	Завдання
1	Добуток найбільшого та найменшого елементів масиву a
2	Сума елементів масиву a, значення яких двозначні парні числа
3	Сума від'ємних елементів масиву, що діляться на 3
4	Найбільший елемент масиву з непарним номером
5	Найменший елемент масиву з парним номером
6	Сума квадратів елементів масиву з парними номерами
7	Сума елементів масиву a, значення яких кратні N
8	Середнє арифметичне першого та найбільшого елементів масива
9	Сума найбільшого та останнього елементів масива
10	Сума елементів, що діляться на 3

Продовж. табл. 8.3

Варіант	Завдання
11	Половина суми елементів масиву з непарними номерами
12	Кількість елементів масиву, що діляться на 7

Контрольні питання

1. Як у C++ описуються одновимірні масиви?
2. Які операції можна виконувати з одновимірними масивами?
3. Для чого в мовах C/C++ використовуються символьні масиви?
4. У чому головна відмінність опису функцій у C++ від опису функцій у Kotlin?
5. Які існують способи передавання параметрів у функцію в мові C++?
6. Що таке перевантаження функцій?
7. Як описати параметри функцій, що можуть використовувати значення «за замовчуванням»?
8. У чому відмінність використання функцій з параметрами «за замовчуванням» у мовах C++ та Kotlin?
9. Що таке рекурсія?
10. Чи завжди рекурсивний алгоритм може бути перетворений у циклічний?
11. Назвіть види рекурсії.
12. У чому полягає особливість хвостової рекурсії?

Тема 9. РОБОТА З ФАЙЛАМИ

Більшість комп'ютерних програм працюють з файлами, і тому виникає необхідність створювати, видаляти, записувати, читати, відкривати файли.

Що ж таке файл? *Файл* – іменований набір байтів, який може бути збережений на деякому накопичувачі. Тепер зрозуміло, що під файлом мається на увазі деяка послідовність байтів, яка має своє, унікальне ім'я, наприклад, file.txt. В одній директорії не можуть знаходитися файли з однаковими іменами. Під іменем файлу розуміється не тільки його назва, а й розширення, наприклад: file.txt та file.dat – різні файли, хоч і мають однакові назви.

Існує таке поняття, як повне ім'я файлу – це повна адреса до директорії файлу із зазначенням імені файлу, наприклад: `D:\docs\file.txt`. Важливо розуміти ці базові поняття, інакше складно буде працювати з файлами.

Файли в мові C++

Для роботи з файлами необхідно підключити заголовний файл `<fstream>`. У `<fstream>` визначені кілька класів і підключені заголовні файли `<ifstream>` – файлове введення та `<ofstream>` – файлове виведення.

Файлове введення / виведення аналогічне стандартному введенню / виведенню, єдина відмінність – це те, що введення / виведення виконаються не на екран, а у файл. Якщо введення / виведення на стандартні пристрої виконується

за допомогою об'єктів `cin` і `cout`, то для організації файлового введення / виводу досить створити власні об'єкти, які можна використовувати аналогічно операторам `cin` і `cout`.

Наприклад, необхідно створити текстовий файл і записати в нього рядок:

```
Робота з файлами в C++.
```

Для цього необхідно виконати наступні кроки:

- створити об'єкт класу `ofstream`;
- зв'язати об'єкт класу з файлом, у який проводитиметься запис;
- записати рядок у файл;
- закрити файл.

Чому необхідно створювати об'єкт класу `ofstream`, а не класу `ifstream`? Тому, що потрібно зробити запис даних у файл, а якби потрібно було зчитувати дані з файлу, то створювався б об'єкт класу `ifstream`.

```
// створюємо об'єкт для запису у файл
ofstream /*ім'я об'єкта*/; // об'єкт класу ofstream
```

Назвемо об'єкт – `fout`. Ось що вийде:

```
ofstream fout;
```

Для чого нам об'єкт? Об'єкт необхідний, щоб можна було виконувати запис у файл. Тепер уже об'єкт створений, але не пов'язаний з файлом, у який потрібно записати рядок.

```
fout.open("lab3cpp.txt"); // зв'язуємо об'єкт з файлом
```

Через операцію крапка отримуємо доступ до методу класу `open()`, у круглих дужках якого вказуємо ім'я файлу. Зазначений файл буде створений у поточній директорії програми. Якщо файл з таким іменем існує, то він буде замінений новим. Отже, файл відкритий, залишилося записати в нього потрібний рядок. Робиться це так:

```
fout << "Робота з файлами у C++"; // записуємо рядок
у файл
```

Використовуючи операцію передачі в потік спільно з об'єктом `fout`, рядок `Робота з файлами у C++` записується у файл. Так як більше немає необхідності змінювати вміст файлу, його треба закрити, тобто відокремити об'єкт від файлу.

```
fout.close(); // закриваємо файл
```

Підсумок – створений файл з рядком `Робота з файлами у C++`

Примітка. Кроки 1 і 2 можна об'єднати, тобто в одному рядку створити об'єкт і пов'язати його з файлом. Робиться це так:

```
ofstream fout("lab3cpp.txt"); // створюємо об'єкт класу ofstream та зв'язуємо його з файлом lab3cpp.txt
```

Об'єднаємо весь код і отримаємо таку програму:

```
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("lab3cpp.txt"); // створюємо об'єкт класу ofstream та зв'язуємо його з файлом lab3cpp.txt
    fout << "Робота з файлами у C++"; // запис рядку у файл
    fout.close(); // закриваємо файл
    return 0;
}
```

Залишилося перевірити правильність роботи програми, а для цього відкриваємо файл `lab3cpp.txt` і дивимося його вміст, повинно бути – `Робота з файлами у C++`.

Для того щоб прочитати файл, знадобиться виконати ті ж кроки, що і під час запису у файл з невеликими змінами:

1. Створити об'єкт класу `ifstream` і пов'язати його з файлом, з якого буде проводитися зчитування.
2. Прочитати файл.
3. Закрити файл.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    char buff[50]; // буфер проміжного зберігання тек-
сту, що зчитується з файлу
    ifstream fin("lab3cpp.txt"); // відкрили файл для
читання
    fin >> buff; // прочитали перше слово з файлу  (*)
    cout << buff << endl; // вивели на екран це слово
    fin.getline(buff, 50); // прочитали рядок з файлу (**)
    fin.close(); // закриваємо файл
    cout << buff << endl; // надрукували цей рядок
    return 0;
}
```

У програмі показані два способи читання з файлу, перший – використовуючи операцію передачі в потік, другий – використовуючи функцію `getline()`. У першому випадку зчитується тільки перше слово, а в другому випадку зчитується рядок, довжиною 50 символів. Але оскільки у файлі залишилося менше 50 символів, то зчитуються символи включно до останнього. Зверніть увагу на те, що зчитування вдруге (рядок помічений **) продовжилося, після першого слова, а не з початку, так як перше слово було прочитано в рядку поміченому *.

Результат роботи програми показаний нижче:

```
Робота
з файлами в C++
```

Програма спрацювала правильно, але не завжди так буває, навіть у тому випадку, якщо з кодом усе в порядку. Наприклад, у програму передано ім'я неіснуючого файлу або в імені допущена помилка. Що тоді? У цьому випадку нічого не відбудеться взагалі. Файл не буде знайдений, а значить і прочитати його неможливо. Тому програма проігнорує рядки, де виконується робота з файлом.

У результаті коректно завершиться робота програми, але нічого на екрані показано не буде. Здавалося б це цілком нормальна реакція на таку ситуацію. Але простому користувачеві не буде зрозуміло, у чому справа і чому на дисплеї не з'явився рядок з файлу. Так ось, щоб усе було максимально зрозуміло, в C++ передбачена така функція – `is_open()`, яка повертає цілі значення: 1 – якщо файл був успішно відкритий, 0 – якщо файл відкритий не був.

Доопрацюємо програму з відкриттям файлу, таким чином, щоб у випадку, якщо файл не відкритий, виводилося відповідне повідомлення.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    char buff[50]; // буфер тимчасового зберігання
тексту, що читається з файлу
    ifstream fin("lab3cpp.doc"); // (ВВЕЛИ НЕКОРЕКТНЕ
ІМ'Я ФАЙЛУ)
    if (!fin.is_open()) // якщо файл не выдкрито
        cout << "Файл не може бути відкрито!\n"; //
повідомити про це
    else
    {
    fin >> buff; // прочитали перше слово з файлу
    cout << buff << endl; // вивели це слово
    fin.getline(buff, 50); // прочитали рядок з файлу
    fin.close(); // закриваємо файл
    cout << buff << endl; // виводимо цей рядок
    }
    return 0;
}
```

Результат роботи програми показаний нижче:

```
Файл не може бути відкрито!
```

Як видно з результату роботи програми, вона повідомила про неможливість відкрити файл. Тому, якщо про-

грама працює з файлами, рекомендується використовувати цю функцію, `is_open()`, навіть, якщо впевнені, що файл існує.

Режими відкриття файлів

Режими відкриття файлів установлюють характер використання файлів. Для установки режиму у класі `ios_base` передбачені константи, які визначають режим відкриття файлів (табл. 9.1).

Таблиця 9.1. **Константи, що визначають режим відкриття файлів**

Константа	Опис
`ios_base::in`	відкрити файл для читання
`ios_base::out`	відкрити файл для запису
`ios_base::ate`	під час відкриття перемістити покажчик у кінець файлу
`ios_base::app`	відкрити файл для запису в кінець файлу
`ios_base::trunc`	видалити вміст файлу, якщо він існує
`ios_base::binary`	відкриття файлу у двійковому режимі

Режими відкриття файлів можна встановлювати безпосередньо під час створення об'єкта або виконання функції `open()`

```
// відкриття файлу для додавання інформації в кінець файлу
ofstream fout("lab3cpp.txt", ios_base::app);
// відкриття файлу для додавання інформації в кінець файлу
fout.open("lab3cpp.txt", ios_base::app);
```

Режими відкриття файлів можна комбінувати за допомогою порозрядної логічної операції `або |`, наприклад: `ios_base::out | ios_base::trunc` – відкриття файлу для запису, попередньо очистивши його.

Об'єкти класу `ofstream`, під час зв'язки з файлами за замовчуванням містять режими відкриття файлів `ios_base::out | ios_base::trunc`. Тобто файл буде створений, якщо не існує. Якщо ж файл існує, то його вміст буде видалено, а сам файл буде готовий до запису. Об'єкти класу `ifstream`, зв'язуючись з файлом, мають за замовчуванням режим відкриття файлу `ios_base::in` – файл відкритий тільки для читання. Режим відкриття файлу ще називають – «прапорець», для зручності надалі використовується саме цей термін.

Зверніть увагу на те, що прапорці `ate` і `app` за описом дуже схожі, вони обидва переміщують покажчик у кінець файлу, але прапор `app` дозволяє проводити запис, тільки в кінець файлу, а прапор `ate` просто переставляє прапор у кінець файлу і не обмежує місця запису.

Розробимо програму, яка, використовуючи операцію `sizeof()`, буде обчислювати характеристики основних типів даних у C++ і записувати їх у файл.

Характеристики:

• число байт, відводиться під тип даних;

• максимальне значення, яке може зберігати певний тип даних.

```
#include <iostream>
#include <fstream> // робота з файлами
#include <iomanip> // маніпулятори введення/виведення
using namespace std;
int main()
{
    // зв'язуємо об'єкт з файлом, при цьому файл від-
криваємо в режимі запису, попередньо видаливши всі
дані з нього
    ofstream fout("data_types.txt", ios_base::out |
ios_base::trunc);
    if (!fout.is_open()) // якщо файл не було відкрито
    {
```

```
     cout << "Файл не може бути відкритим або створе-
ним\n";
     return 1;
    }
        fout << "     data type      " << "byte" <<
"       "
             << "    max value  " << endl // заголовки
стовпців
             << "bool               =  " <<
sizeof(bool) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних
bool*/
            << (pow(2,sizeof(bool) * 8.0) - 1) << endl
             << "char               =  " <<
sizeof(char) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних char*/
            << (pow(2,sizeof(char) * 8.0) - 1) << endl
             << "short int          =  " <<
sizeof(short int) << "    "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних short
int*/
             << (pow(2,sizeof(short int) * 8.0 - 1) - 1)
<< endl
             << "unsigned short int =  " <<
sizeof(unsigned short int)  << "          "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних
unsigned short int*/
             << (pow(2,sizeof(unsigned short int) *
8.0) - 1) << endl
             << "int                =  " << sizeof(int)
<< "          "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних int*/
             << (pow(2,sizeof(int) * 8.0 - 1) - 1) <<
endl
```

```
            << "unsigned int       =  " <<
sizeof(unsigned int) <<
"         "  << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних
unsigned int*/
            << (pow(2,sizeof(unsigned int) * 8.0) - 1)
<< endl
            << "long int           =  " <<
sizeof(long int) <<
"         "  << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних long
int*/
            << (pow(2,sizeof(long int) * 8.0 - 1) - 1)
<< endl
            << "unsigned long int  =  " <<
sizeof(unsigned long int) << "         "
            << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних long
int*/
            << (pow(2,sizeof(unsigned long int) *
8.0) - 1) << endl
           << "float              =  " << sizeof(float) <<
"         "  << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних float*/
            << (pow(2,sizeof(float) * 8.0 - 1) - 1) <<
endl
            << "long float         =  " << sizeof(long
float) <<
"         "  << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних long
float*/
            << (pow(2,sizeof(long float) * 8.0 - 1) - 1)
<< endl
            << "double             =  " <<
sizeof(double) <<
"         "  << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних double*/
            << (pow(2,sizeof(double) * 8.0 - 1) - 1)
<< endl;
        fout.close();
```

```
// програма більше не використовує файл, тому його
треба закрити
    cout << "Дані успішно записані у файл data_types.
txt\n";
    return 0;
}
```

Можна помітити, що стандартне введення / виведення і файлове введення / виведення використовуються абсолютно аналогічно. У кінці програми ми явно закрили файл, хоча це й не обов'язково під час використання деяких компіляторів, але вважається хорошим тоном програмування та необхідно для сумісності з різноманітними версіями компіляторів C++. Варто відзначити, що всі функції та маніпулятори використовувані для форматування стандартного введення / виведення актуальні і для файлового введення / виведення.

Файли в мові Kotlin

Запис у файл у Kotlin

Kotlin пропонує різні способи запису у файл у вигляді методів розширення для `java.io.File`.

Ми використаємо кілька з них, щоб продемонструвати різні способи запису у файли в Kotlin:

- `writeText` – дозволяє записувати у файл напряму з рядків String;
- `writeBytes` – використовується для запису напряму з масивів байтів `ByteArray`;
- `printWriter` – надає нам об'єкт `PrintWriter` із широкими можливостями;
- `bufferedWriter` – надає можливість запису з використанням об'єкта `BufferedWriter`.

Запис «напряму»

`writeText`

Можливо, найпростіший метод розширення, `writeText` приймає вміст як аргумент `String` і записує його безпосе-

редньо у вказаний файл. Даний уміст записується з використанням кодування у форматі UTF-8 (за замовчуванням) або будь-яким іншим указаним кодуванням:

```
File(fileName).writeText(fileContent)
```

Цей метод внутрішньо викликає writeBytes, як буде описано нижче. Але спочатку він перетворює заданий уміст у масив байтів, використовуючи вказане кодування writeBytes.

Метод writeBytes бере аргумент типу ByteArray і безпосередньо записує його у вказаний файл. Це корисно, коли ми маємо вміст як масив байтів, а не як звичайний текст.

```
File(fileName).writeBytes(fileContentAsArray)
```

Якщо даний файл існує, він перезаписується.

Запис у файл за допомогою Writers

Kotlin також пропонує методи розширення, які надають нам об'єкт типу Java Writer.

printWriter

Якщо ми хочемо використовувати Java PrintWriter, Kotlin надає функцію printWriter саме для цієї мети. За допомогою нього ми можемо виводити відформатовані представлення об'єктів у вихідний потік (OutputStream):

```
File(fileName).printWriter()
```

Ця функція (метод) повертає новий екземпляр PrintWriter. Далі ми можемо скористатися використанням методу use для роботи з ним:

```
File(fileName).printWriter().use { out -> out.
println(fileContent) }
```

За допомогою методу (функції) use ми можемо виконати функцію на ресурсі, яка закривається після завершення. Ресурс (у даному випадку – файл) буде закрито незалеж-

но від того, успішно виконана функція чи вона призвела до помилки.

`bufferedWriter`

Подібним чином `Kotlin` також надає функцію `bufferedWriter`, яка надає нам об'єкт `Java BufferedWriter`.

```
File(fileName).bufferedWriter()
```

Подібно до `PrintWriter`, ця функція повертає новий екземпляр `BufferedWriter`, який пізніше ми можемо використовувати для запису вмісту файлу.

```
File(fileName).bufferedWriter().use { out -> out.
write(fileContent) }
```

Розглянемо приклад запису у файл повідомлення `"Hello, Kotlin!"`

```
import java.io.File

fun main() {
val writer = File("lab3kt.txt").printWriter() writer.
println("Hello,  Kotlin!") writer.close()
}
```

або з використанням функції `use`:

```
import java.io.File

fun main() {
File("lab3kt.txt").printWriter().use                {
it.println("Hello, Kotlin!")
}
}
```

Читання з файлу в Kotlin

Насамперед створимо вхідний файл, який буде читати програма мовою Kotlin. Ми створюємо файл під назвою example.txt і зберігаємо його в каталог, до якого можна отримати доступ за допомогою нашого коду.

Уміст файлу може бути, наприклад, таким:

```
Hello from Kotlin! It's:
1. Concise
2. Safe
```

3. Interoperable
4. Tool-friendly

Розглянемо різні способи, за допомогою яких ми можемо прочитати цей файл. Ми повинні передати повний шлях до файлу, створеного вище, як вхідні дані для таких методів, або відносний шлях щодо розміщення файлу програми.

forEachLine

Читає файл рядок за рядком, використовуючи вказану кодову таблицю (за замовчуванням UTF-8), і викликає дію для кожного рядка:

```
File(fileName).forEachLine { println(it) }
```

У цьому прикладі програма виводить на екран уміст файлу «рядок за рядком». Замість `println(it)` може бути будь-яка дія, чи декілька дій, де `it` – рядок, що прочитаний з файлу.

useLines

Викликає вказану операцію «зворотного виклику» у блоці, надаючи йому послідовність усіх рядків у файлі.

Після завершення обробки файл закривається:

```
val listOfLines : List<String> = File(fileName).
useLines { it.toList() }
```

Таким чином, елементами списку будуть рядки вхідного файлу

bufferedReader

Повертає новий об’єкт `BufferedReader` для читання вмісту файлу. Отримавши `BufferedReader`, ми можемо прочитати всі рядки в ньому:

```
val listOfLines : List<String> = File(fileName).
bufferedReader().readLines()
```

Також, використовуючи `BufferedReader`, можна організувати «більш традиційний» процес обробки файлу в циклі.

```
var line: String?
while (reader.readLine().also { line = it } != null)
```

```
{ println(line)
}
reader.close()
```

або з використанням `use`:

```
var line: String? File("lab3kt.txt").bufferedReader().
use { reader ->
while (reader.readLine().also { line = it } != null)
{ println(line)
}
}
```

`readLines`

Безпосередньо читає вміст файлу у список рядків:

```
val listOfLines : List<String> = File(fileName).
readLines()
```

Примітка. Цей метод не рекомендується використовувати для величезних файлів.

`inputStream`

Створює новий об'єкт `FileInputStream` для файлу й повертає його в результаті.

Отримавши вхідний потік, ми можемо перетворити його в байти, а потім у звичайний рядок:

```
val s : String = File(fileName).inputStream().
readBytes().toString(Charsets.UTF_8)
```

`readText`

Зчитує весь уміст файлу як рядок із зазначеним кодуванням (за замовчуванням UTF-8):

```
val s : String = File(fileName).readText(Charsets.
UTF_8)
```

Примітка. Цей метод не рекомендується використовувати для величезних файлів і має внутрішнє обмеження у 2 ГБ.

`Scanner`

Для введення даних, як з клавіатури, так і з файлу можна використовувати клас `Scanner` зі стандартної бібліотеки `Java`.

```
val scanner = Scanner(File(fileName))
val intValue = scanner.nextInt() // зчитування цілого числа
val doubleValue = scanner.nextDouble() // зчитування дійсного числа типу Double
// ... аналогічно для інших типів даних
```

Лабораторний практикум № 9

Мета: набути навичок створення та реалізації програм, що реалізують операції введення-виведення із файлами.

Завдання

Для виконання завдань 9.1 та 9.2 розробіть по дві програми, що розв'язують ці завдання: мовою C++ та мовою Kotlin. Порівняйте зручність інструментів цих мов програмування, що використовуються для роботи з файлами.

Завдання 9.1.

Розробити програму, що виконує обчислення з табл. 9.2, використовуючи дані, що знаходяться у файлі.

У всіх варіантах дано файл `f.txt`, компоненти якого – дійсні числа. (*Файл підготувати самостійно. Кількість елементів у файлі не менше 20*).

Таблиця 9.2.

Варіант	Завдання
1	Знайти суму компонентів файлу
2	Знайти добуток компонентів файлу
3	Знайти суму квадратів компонентів файлу
4	Знайти модуль суми та квадрат добутку компонентів файлу

Продовж. табл. 9.2

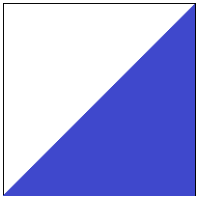
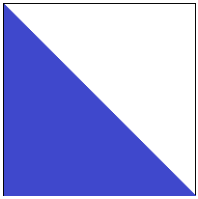
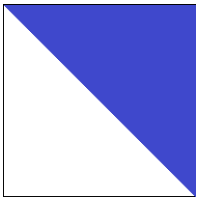
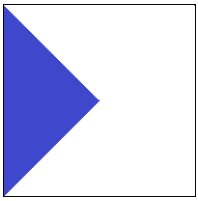
Варіант	Завдання
5	Знайти останній компонент файлу
6	Знайти найменше значення серед компонентів файлу
7	Знайти найменше значення серед компонентів файлу з парними номерами
8	Знайти найбільше значення серед компонентів файлу з непарними номерами
9	Знайти суму найбільшого та найменшого компонентів файлу
10	Знайти різницю першого та останнього компонентів файлу
11	Знайти суму компонентів файлу з непарними номерами
12	Знайти суму від'ємних компонентів файлу
13	Знайти кількість від'ємних компонентів з непарними номерами
14	Знайти суму компонентів, ціла частина яких кратна 3
15	Знайти суму компонентів між першим та останнім від'ємними
16	Знайти кількість компонентів у файлі
17	Знайти суму першого від'ємного та останнього додатного компонентів файлу
18	Знайти добуток усіх від'ємних компонентів файлу
19	Знайти суму квадратів усіх компонентів файлу
20	Знайти середнє арифметичне квадратів тих компонентів файлу, які не дорівнюють нулю

Завдання 9.2.

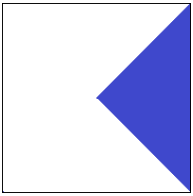
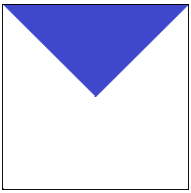
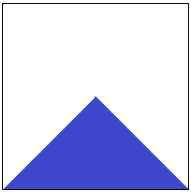
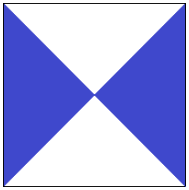
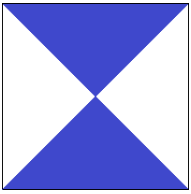
Розробити програму, що зчитує з файлу z32.txt елементи двовимірного масиву цілих чисел розміром NxN елементів та опрацьовує його згідно з варіантом.

Примітка. Перший рядок файлу містить одне ціле число N. Наступні N рядків містять по N дійсних чисел – елементи заданого масиву. Файл можна отримати у викладача, або завантажити за посиланням: https://github.com/kafedra-iust/lab3cpp .

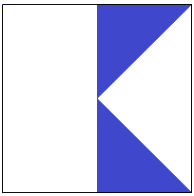
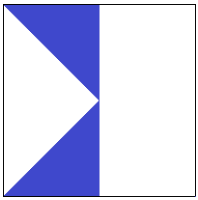
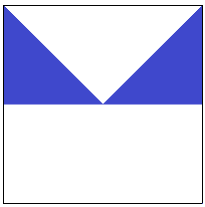
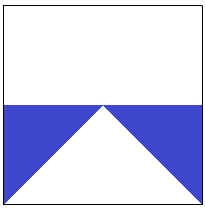
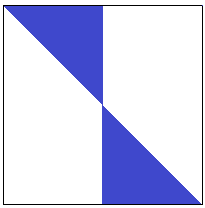
Таблиця 9.3.

Варіант	Завдання	
1		Знайти суму елементів заштрихованої частини
2		Знайти індекси і значення найбільшого елемента заштрихованої частини
3		Знайти суму модулів елементів заштрихованої частини
4		Знайти індекси і значення найменшого елемента заштрихованої частини
5		Знайти суму від’ємних елементів заштрихованої частини

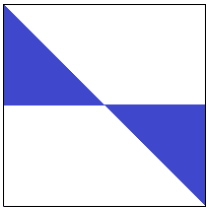
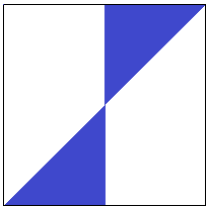
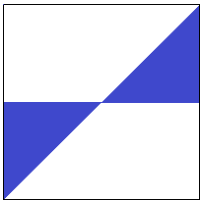
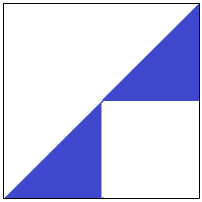
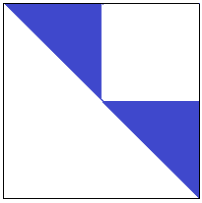
Продовж. табл. 9.3

Варіант	Завдання	
6		Знайти індекси і значення найбільшого парного елемента заштрихованої частини
7		Знайти суму додатних елементів заштрихованої частини
8		Знайти суму непарних елементів заштрихованої частини
9		Знайти кількість від'ємних елементів та суму додатних елементів заштрихованої частини
10		Знайти суму непарних додатних елементів заштрихованої частини

Продовж. табл. 9.3

Варіант	Завдання	
11		Знайти індекси найбільшого від'ємного та найменшого додатного елементів заштрихованої частини
12		У заштрихованій частині знайти кількість елементів, що відрізняються від найменшого елемента не більше ніж на 10 %
13		Знайти суму елементів заштрихованої частини, що діляться на 3
14		У заштрихованій частині знайти кількість елементів, що відрізняються від найбільшого елемента не більше ніж на 10 %
15		Знайти різницю найбільшого та найменшого елементів у заштрихованій частині

Продовж. табл. 9.3

Варіант	Завдання	
16		Знайти суму елементів заштрихованої частини, кратних 5
17		Обчислити середнє арифметичне від'ємних елементів заштрихованої частини
18		Обчислити середнє арифметичне додатних елементів заштрихованої частини
19		Знайти суму логарифмів модулів елементів заштрихованої частини
20		Знайти суму квадратів елементів заштрихованої частини

Контрольні питання

1. Що таке файл?
2. Які формати файлів існують?
3. Який порядок роботи з файлами?
4. Які режими роботи з файлами в C++ існують?
5. Як відбувається читання даних у текстовому форматі мовою C++?
6. Як відбувається читання даних у текстовому форматі мовою Kotlin?
7. Як відбувається запис даних у текстовому форматі мовою C++?
8. Як відбувається запис даних у текстовому форматі мовою Kotlin?
9. Як відбувається читання та запис даних у бінарному форматі мовою C++?
10. Як відбувається читання та запис даних у бінарному форматі мовою Kotlin?

Тема 10. СТРУКТУРИ (C/C++). DATA CLASSES (KOTLIN)

Загальні відомості C++

Структура – це об'єднання різних змінних (навіть з різними типами даних), якому можна присвоїти ім'я. Наприклад, можна об'єднати дані про об'єкт – Будинок: місто (у якому будинок знаходиться), вулиця, кількість квартир, інтернет (проведено чи ні) і т. д. в одній структурі.

Загалом можна зібрати в одну сукупність дані про все, що завгодно, точніше про все, що необхідно конкретному програмісту.

Розглянемо простий приклад, який допоможе ознайомитися зі структурами і покаже, як з ними працювати. У цій програмі ми створимо структуру, створимо об'єкт структури, заповнимо значеннями елементи структури (дані про об'єкт) і виведемо ці значення на екран.

```
#include <string>
#include <iostream>

using namespace std;

struct home     // Створюємо структуру
{
    string owner;      // тут зберігатиметься ім'я
власника
    string city;       // назва міста
    int amountRooms;   // кількість кімнат
```

```
    double price;        // ціна
};

int main()
{
    home apartment1;   // об'єкт структури з типом
даних, іменем структури home

    apartment1.owner = "Сєня"; // заповнюємо дані
про власника і т.д.
    apartment1.city = "Миколаїв";
    apartment1.amountRooms = 5;
    apartment1.price = 150000;

    cout << "Власник квартири: " << apartment1.owner
<< endl;
    cout << "Квартира знаходиться в місті: " <<
apartment1.city << endl;
    cout << "Кількість кімнат: " << apartment1.
amountRooms << endl;
    cout << "Ціна: " << apartment1.price << " $" <<
endl;

    return 0;
}
```

У результаті роботи цієї програми ми побачимо таке:

```
Власник квартири: Сєня
Квартира знаходиться в місті: Миколаїв
Кількість кімнат: 5
Ціна: 150000 $
```

Зауважимо, що за сучасним стандартом мови програмування C++ простим змінним бажано надавати початкові значення, тому опис структури, наведеної вище, краще переписати, наприклад, таким чином:

```
struct home     // Створюємо структуру
{
    string owner;      // тут зберігатиметься ім'я
власника
```

```
    string city;        // назва міста
    int amountRooms{1};   // кількість кімнат - за
замовчуванням завжди є одна кімната
    double price{};       // ціна - за замовчуванням
дорівнює 0 (0 можна не писати в дужках)
};
```

Об'єкт структури можна оголосити до функції `main()`. Це виглядало б так:

```
struct home
{
    string owner;
    string city;
    int amountRooms{1};
    double price{};
} apartment1;
```

Проте, як було відмічено в попередніх роботах, оголошувати глобальні змінні варто з обережністю (а краще, узагалі уникати цього). Ініціалізувати структуру можна і таким способом:

```
home apartment2 = {"Сєня", "Миколаїв", 5, 15000};
```

але так роблять украй рідко.

Структуру можна вкладати в інші структури. Це ми розглянемо в наступному прикладі:

```
#include <string>
#include <iostream>

using namespace std;

struct date //створюємо ще одну структуру, щоб
вкласти її у структуру home // дата, коли побудували
дім
{
    string month;   // Місяць побудови дому
    int year{1900}; // рік
};

struct home     // Створюємо структуру
{
```

```
    string owner;      // тут зберігатиметься ім'я
власника
    string city;       // назва міста
    int amountRooms{1};   // кількість кімнат
    double price{};       // ціна
    date built; // вкладаємо одну структуру у визна-
чення другої
};

void show(home obj) // створюємо функцію, яка приймає
структуру, як параметр
{
    cout << "Власник квартири: "           << obj.
owner << endl;
    cout << "Квартира знаходиться в місті: " << obj.
city << endl;
    cout << "Кількість кімнат: "     << obj.
amountRooms << endl;
    cout << "Ціна: "               << obj.price << "
$" << endl;
    cout << "Дата будівництва: "             << obj.
built.month << ' ' << obj.built.year << endl;
}

void showByPtr(const home *pApartment) {
    // Зверніть увагу, як треба звертатись до еле-
мента структури через вказівник
    // використовуємо оператор  ->
    cout << "Власник квартири: "             <<
pApartment->owner << endl;
    cout << "Квартира знаходиться в місті: " <<
pApartment->city << endl;
    cout << "Кількість кімнат: "             <<
pApartment->amountRooms << endl;
    cout << "Ціна: "                         <<
pApartment->price << " $" << endl;
    cout << "Дата будівництва: "             <<
pApartment->built.month << ' ' << pApartment->built.
year << endl;
}
```

```
int main()
{
    home apartment1;   // об'єкт структури з типом
даних, іменем структури home

    apartment1.owner = "Сєня"; // заповнюємо дані
про власника і т.д.
    apartment1.city = "Миколаїв";
    apartment1.amountRooms = 5;
    apartment1.price = 150000;
    apartment1.built.month = "січень";
    apartment1.built.year = 2013;

    show(apartment1);

    struct home *pApartment; // це вказівник на
структуру (вказівники докладно розглядатимуться у
наступних лабораторних роботах)
    pApartment = &apartment1;

    showByPtr(pApartment); // передаємо вказівник на
структуру, як параметр

    home apartment2; // створюємо та заповнюємо дру-
гий об'єкт структури

    apartment2.owner = "Вова";
    apartment2.city = "Київ";
    apartment2.amountRooms = 17;
    apartment2.price = 3000000;
    apartment2.built.month = "березень";
    apartment2.built.year = 2019;

    home apartment3 = apartment2; // створюємо тре-
тій об'єкт структури та присвоюємо йому дані об'єкта
apartment2

    show(apartment3);

    return 0;
}
```

Зауважимо, що згідно з новими стандартами мови C++, функція, що отримує параметром стуктуру може бути оптимізована з використанням параметра типу «константне посилання». Рекомендацію щодо цього можна побачити в середовищі CLion (рис. 10.1).

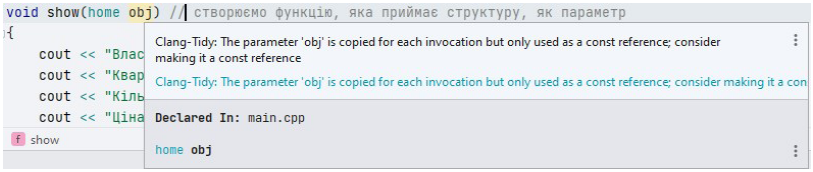


Рис. 10.1. Рекомендація щодо використання параметра типу «константне посилання» в CLion

Після такої зміни функція буде мати такий вигляд:

```
void show(const home& obj) // створюємо функцію, яка
приймає структуру, як параметр
{
    cout << "Власник квартири: "              << obj.
owner << endl;
    cout << "Квартира знаходиться в місті: " << obj.
city << endl;
    cout << "Кількість кімнат: "           << obj.
amountRooms << endl;
    cout << "Ціна: "                      << obj.
price << " $" << endl;
    cout << "Дата будівництва: "             << obj.
built.month << ' ' << obj.built.year << endl;
}
```

Запис структур у файл / Читання структур з файлів

Раніше ми розглядали лише текстові файли.

Текстові файли:

- містять текстове представлення інформації;
- для запису / читання потрібно перетворювати;
- можна переглядати / читати в текстовому редакторі.

Однак дуже часто використовується інший вид файлів – бінарні файли.

Бінарні файли:

- містять дані в тому ж форматі, як у пам'яті;
- не потрібно перетворень;
- для перегляду / читання потрібна програма.

Для запису структур у текстовий файл можна використовувати такі ж самі засоби, як і для виведення на екран. У попередній лабораторній роботі було розглянуто роботу з текстовими файлами – вона використовує ті ж самі операції, що й виведення на екран.

Щоб записати структуру в бінарний файл, потрібно знати розміри структури.

Якщо в полях структури є покажчики, то правильний розмір структури дізнатися не вийде.

Щоб записати структуру у файл, потрібно повідомити компілятору:

- адресу структури, приведену до типу «покажчик на char»;
- розмір записуваної структури.

```
// apartment1 - структура, яку потрібно записати
у файл
ofstream fout("f1.dat", ios::binary); // відкриваємо
файл, вказуючи прапорець "бінарне введення-виведення"
fout.write((char*)&apartment1, sizeof(home)); // за-
писуємо структуру у файл
fout.close();
```

Якщо потрібно записати у файл масив структур, можна скористатися тим фактом, що у C++ масив – це фактично вказівник на його перший елемент.

```
// ArrX - масив структур , який потрібно записати
у файл
// n - кількість елементів у масиві, що треба записати
у файл
int n = 3;
```

```
home ArrX[] = {apartment1, apartment2, apartment3};
ofstream fout("fn.dat", ios::binary); // відкриваємо
файл, вказуючи прапорець "бінарне введення-виведення"
fout.write((char*)&ArrX[0], sizeof(home)*n); // запи-
суємо масив структур у файл
fout.close();
```

Аналогічно відбувається читання структур з файлу

```
int n = 3;
home ArrY[n];
ifstream  fin("f1.txt",ios::binary);  //  відкриваємо
файл, вказуючи прапорець "бінарне введення-виведення"
fin.read((char*)&ArrY[0], sizeof(home)*n);
fin.close();
```

Загальні відомості Kotlin

Концепція структур у мові Kotlin відсутня. Найбільш близькою за сенсом є концепція класів даних – data class.

Розглянемо приклад створення та опрацювання таких даних аналогічно тим, що були використані в попередніх розділах роботи.

Опис класу даних відбувається таким чином:

```
data class Home(
    var owner: String,
    var city: String,
    var amountRooms: Int,
    var price: Double
)
```

Уся програма, що використовує такий клас даних, може мати такий вигляд:

```
fun main() {
    // створюємо об'єкт типу Home та заповнюємо даними
    val apartment1 = Home("Сєня", "Миколаїв", 5,
150_000.0)

    // виводимо вміст об'єкта
    println("Власник квартири: ${apartment1.owner}")
    println("Квартира знаходиться в місті:
${apartment1.city}")
```

```
    println("Кількість кімнат: ${apartment1.
amountRooms}")
    println("Ціна: ${apartment1.price} $")
    println()

    // до полів об'єкта можна звертатися і так:
    with(apartment1) {
        println("Власник квартири: $owner")
        println("Квартира знаходиться в місті:
$city")
        println("Кількість кімнат: $amountRooms")
        println("Ціна: $price $")
    }
}

data class Home(
    var owner: String,
    var city: String,
    var amountRooms: Int,
    var price: Double
)
```

Зверніть увагу, що клас даних Home може бути оголошеним як до функції, що його використовує, так і після неї.

Іноді, для зручності, такі класи розміщують в окремих файлах з іменем, що співпадає з іменем класу та має розширення `.kt`.

Поле класу даних також може мати тип, що оголошений як інший клас даних:

```
fun main() {
    // створюємо об'єкт типу Home та заповнюємо даними
    val apartment1 = Home("Сєня", "Миколаїв", 5,
150_000.0, Date("січень", 2013))

    show(apartment1)

    // створюємо другий об'єкт типу Home
    val apartment2 = Home("Вова", "Київ", 17,
3_000_000.0, Date("березень", 2019))
```

```
    // копіюємо дані другого об'єкта у третій
    val apartment3 = apartment2.copy()

    // створюємо додаткове посилання на об'єкт
apartment2. Обидва посилання працюють з одним й тим
самим об'єктом
    val apartment4 = apartment2
    apartment2.price /= 1000 // зменшуємо ціну
apartment4

    println("apt 4 price = ${apartment4.price}") //
можна побачити, що ціна змінилась також
    println("apt 3 price = ${apartment3.price}") //
можна побачити, що ціна не змінилась
}

fun show(apartment: Home) {
    // виводимо вміст об'єкта
    println("Власник квартири: ${apartment.owner}")
    println("Квартира знаходиться в місті:
${apartment.city}")
    println("Кількість кімнат: ${apartment.
amountRooms}")
    println("Ціна: ${apartment.price} $")
    println()
}

// клас даних Home
data class Home(
    var owner: String,
    var city: String,
    var amountRooms: Int,
    var price: Double,
    var built: Date
)

// клас даних для дати, що буде полем в класі Home
data class Date(var month:String, var year:Int=1990)
```

Зауважимо, що кожен об'єкт будь-якого класу даних має вбудовану функцію `copy()`, яка виконує копіювання даних об'єкта в інший об'єкт.

Запис об'єктів класів даних у бінарний (або структурований) файл

У програмах мовою Kotlin для запису бінарних даних ми можемо використовувати механізм «серіалізації». Цей механізм було запозичено з мови програмування Java. Але Kotlin доповнений можливостями вбудованої серіалізації у формати текстових файлів (JSON, XML), що є кросплатформовими та можуть читатися та записуватися різними програмами, які написані будь-якими мовами програмування.

Для запису бінарних даних можна використовувати клас `ObjectOutputStream`:

```
fun writeListToFile(list: List<Home>) {
    ObjectOutputStream(FileOutputStream("houses.
dat")).use {
        it.writeObject(list)
    } // при такому використанні, файл буде автома-
тично закрито після виходу з блоку
}
```

Зверніть увагу, що записувати можна не лише прості об'єкти, а навіть списки об'єктів.

Також, якщо під час описання класу даних, планується, що об'єкти цього класу будуть записуватись у файли, треба в явному вигляді це показати, указавши ім'я «маркерного» інтерфейсу `Serializable` після знака «двокрапка» в рядку оголошення класу:

```
// клас даних Home
data class Home(
    var owner: String,
    var city: String,
    var amountRooms: Int = 1,
```

```
    var price: Double,
    var built: Date
) : Serializable

// клас даних для дати, що буде полем в класі Home
data class Date(var month:String, var year:Int=1990)
: Serializable
```

Для читання даних, що були записані за допомогою `ObjectOutputStream`, слід використовувати `ObjectInputStream`. Це можна зробити, наприклад, так:

```
fun readListFromFile(): List<Home> {
    ObjectInputStream(FileInputStream("houses.
dat")).use {
        return it.readObject() as List<Home>
    }
}
```

Зверніть увагу на конструкцію `as List<Home>`, що вказує компілятору, що об'єкт, який буде прочитано з файлу, треба вважати об'єктом типу `"List<Home>"` – тобто, у цьому випадку, списком об'єктів типу Home.

Лабораторний практикум № 10

Мета: набути навичок створення та реалізації програм, що використовують структури (data class).

Завдання

Завдання 10.1 (C++)

Створити структуру, специфікація якої наведена нижче. Визначити функції, що створюють масив структур. Задати критерій відбору.

1. **Student:** id, Прізвище, Ім'я, По батькові, Дата народження, Адреса, Телефон, Факультет, Курс, Група.

Створити масив структур. Вивести:

a. Список студентів указаного факультету.

b. Список студентів, які народилися після вказаного року.

c. Список навчальної групи в алфавітному порядку.

2. **Customer:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Номер банківського рахунку.

Створити масив структур. Вивести:

a. Список покупців в алфавітному порядку.

b. Список покупців, у яких номер кредитної картки знаходиться в заданому інтервалі.

c. Список покупців, у яких номер банківського рахунку закінчується на вказану цифру.

3. **Patient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Номер медичної картки, Діагноз.

Створити масив структур. Вивести:

a. Список пацієнтів, які мають указаний діагноз.

b. Список пацієнтів, чий номер медичної картки при діленні на 7 дасть указану остачу.

c. Список пацієнтів, номер медичної карти яких знаходиться в заданому інтервалі.

4. **Abiturient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Оцінки. Створити масив структур. Вивести:

a. Список абітурієнтів, які мають незадовільні оцінки.

b. Список абітурієнтів, у яких сума балів вище заданої.

c. Вибрати вказану кількість n абітурієнтів, які мають найбільшу суму балів.

5. **Book:** id, Назва, Автор(и), Видавництво, Рік видання, Кількість сторінок, Ціна, Тип палітурки.

Створити масив структур. Вивести:

a. Список книг заданого автора.

b. Список книг, що видані вказаним видавництвом.

c. Список книг, що видані після заданого року.

6. **House:** id, Номер квартири, Площа, Поверх, Кількість кімнат, Вулиця, Тип будівлі, Термін експлуатації.

Створити масив структур. Вивести:

a. Список квартир, що мають задану кількість кімнат.

b. Список квартир, що мають указану кількість кімнат і розташованих між указаними поверхами.

c. Список квартир, які мають площу, що більше заданої.

7. **Phone:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Дебет, Кредит, Час міських розмов, Час міжнародних розмов.

Створити масив структур. Вивести:

a. Відомості про абонентів, у яких час міських розмов перевищує вказаний.

b. Відомості про абонентів, які користувалися міжнародним зв'язком.

c. Відомості про абонентів в алфавітному порядку.

8. **Car:** id, Марка, Модель, Рік випуску, Колір, Ціна, Реєстраційний номер. Створити масив структур. Вивести:

a. Список автомобілів заданої марки.

b. Список автомобілів заданої моделі, які експлуатуються більше n років.

c. Список автомобілів указаного року випуску, ціна яких більше вказаної.

9. **Product:** id, Найменування, Тип, Виробник, Ціна, Термін зберігання, Кількість.

Створити масив структур. Вивести:

a. Список товарів заданого найменування.

b. Список товарів заданого найменування, ціна яких не більше заданої.

c. Список товарів, термін зберігання яких більше заданого.

10. **Train:** id, Пункт призначення, Номер поїзда, Час відправлення, Число місць (загальних, плацкарт, купе, люкс).

Створити масив структур. Вивести:

a. Список поїздів, які прямують до заданого пункту призначення.

b. Список поїздів, які прямують до заданого пункту призначення та відправляються після вказаної години.

c. Список поїздів, які відправляються до заданого пункту призначення та мають загальні місця.

Завдання 10.2 (C++)

Створити файл для зберігання даних, що зберігаються в масивах структур, описаних у завданні 10.1. Розробити програму, що зчитує дані з файлу в масив структур та записує дані у файл. Виконати запити, описані в завданні 10.1 з використанням масивів структур, та (якщо це можливо) без них.

Завдання 10.3 (Kotlin)

Виконати завдання 10.1 та 10.2 мовою Kotlin

Контрольні питання

1. Що таке комбінований тип даних – структура?
2. Як описуються структури в мовах C та C++?
3. Як використовуються змінні типу «структура» у C та C++?
4. Як здійснюється запис структур до файлу?
5. Як здійснюється читання структур з файлу?
6. Як описується Data Class у Kotlin?
7. У чому особливість Data Class у Kotlin?
8. Як використовуються змінні типу Data Class у Kotlin?
9. Як здійснюється запис Data Class до файлу?
10. Як здійснюється читання Data Class з файлу?

Тема 11. ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ. ПОКАЖЧИКИ

Основні поняття

Покажчик – змінна, значенням якої є адреса комірки пам'яті. Тобто покажчик посилається на блок даних з області пам'яті, причому на саме його початок. Покажчик може посилатися на змінну або функцію. Для цього потрібно знати адресу змінної або функції. Так ось, щоб дізнатися адресу конкретної змінної в C++, існує унарна операція взяття адреси `&`. Така операція витягує адресу оголошених змінних, для того, щоб його присвоїти покажчикові.

Покажчики використовуються для передачі за посиланням даних, що набагато прискорює процес обробки цих даних (у тому випадку, якщо обсяг даних великий), через те, що їх не треба копіювати, як під час передачі за значенням, тобто, використовуючи ім'я змінної. В основному покажчики використовуються для організації динамічного розподілу пам'яті, наприклад, під час оголошення масиву, не треба буде його обмежувати в розмірі. Адже програміст заздалегідь не може знати, якого розміру потрібен масив тому чи іншому користувачеві, у такому випадку використовується динамічне виділення пам'яті під масив. Будь-який покажчик необхідно оголосити перед використанням, як і будь-яку змінну.

```
//оголошення покажчика
/*тип даних*/  * /*ім'я покажчика*/;
```

Принцип оголошення покажчиків такий само, як і принцип оголошення змінних. Відмінність полягає лише в тому, що перед ім'ям ставиться символ зірочки `*`. Візуально покажчики відрізняються від звичайних змінних тільки одним символом.

Під час оголошення покажчиків компілятор виділяє декілька байт пам'яті, які відводяться в залежності від типу даних для зберігання деякої інформації в пам'яті.

Щоб отримати значення, записане в деякій області, на яке посилається покажчик, потрібно скористатися операцією розіменування покажчика `*`. Необхідно поставити зірочку перед ім'ям, і отримаємо доступ до значення покажчика. Розглянемо програму, яка буде використовувати покажчики.

```
#include <iostream>

using namespace std;

int main()
{
    int var = 123; // ініціалізація змінної var числом 123
    int *ptrvar = &var; // покажчик на змінну var (присвоїли адресу змінної покажчику)
    cout << "&var    = " << &var << endl; // адреса змінної var, що міститься в пам'яті, отримана операцією взяття адреси
    cout << "ptrvar  = " << ptrvar << endl; // адреса змінної var, є значенням покажчика ptrvar
    cout << "var     = " << var << endl; // значення у змінній var
    cout << "*ptrvar = " << *ptrvar << endl; // виведення значення, що міститься у змінній var через покажчик, операцією розіменування покажчика
    return 0;
}
```

У програмуванні заведено додавати до імені покажчика префікс `ptr`, таким чином, отримаємо змістовне ім'я покажчика, та вже зі звичайною змінною такий покажчик не сплутати. Результат роботи програми:

```
&var    = 0x22ff08
ptrvar  = 0x22ff08
var     = 123
*ptrvar = 123
```

Покажчики можна порівнювати не тільки на рівність чи нерівність, адже адреси можуть бути менше або більше один відносно одного. Нижче наведено програму, яка буде порівнювати адреси покажчиків.

```
#include <iostream>
using namespace std;

int main()
{
    int var1 = 123; // ініціалізація змінної var1
числом 123
    int var2 = 99; // ініціалізація змінної var2
числом 99
    int *ptrvar1 = &var1; // покажчик на змінну var1
    int *ptrvar2 = &var2; // покажчик на змінну var2
    cout << "var1    = " << var1 << endl;
    cout << "var2    = " << var2 << endl;
    cout << "ptrvar1 = " << ptrvar1 << endl;
    cout << "ptrvar2 = " << ptrvar2 << endl;
    if (ptrvar1 > ptrvar2) // порівнюємо значення
покажчиків, тобто адреси змінних
        cout << "ptrvar1 > ptrvar2" << endl;
    if (*ptrvar1 > *ptrvar2) // порівнюємо значення
змінних, на які посилаються покажчики
        cout << "*ptrvar1 > *ptrvar2" << endl;

    return 0;
}
```

Результат роботи програми:

```
var1    = 123
var2    = 99
ptrvar1 = 0xffffcc2c
ptrvar2 = 0xffffcc28
ptrvar1 > ptrvar2
*ptrvar1 > *ptrvar2
```

У першому випадку ми порівнювали адреси змінних, і можна помітити, що адреса другої змінної завжди менше адреси першої змінної. Під час кожного запуску програми адреси можуть виділяютися різні. У другому випадку ми порівнювали значення цих змінних, використовуючи операцію розіменування покажчика.

З арифметичних операцій, найчастіше використовуються операції додавання, віднімання, інкремент і декремент, бо за допомогою цих операцій, наприклад, у масивах, обчислюється адреса наступного елемента.

Покажчики на покажчики

Покажчики можуть посилатися на інші покажчики. При цьому в комірках пам'яті, на які будуть посилатися перші покажчики, будуть міститися не значення, а адреси інших покажчиків. Число символів `*` під час оголошення покажчика показує порядок покажчика.

Щоб отримати доступ до значення, на яке посилається покажчик, його необхідно розіменовувати відповідну кількість разів. Розглянемо програму, яка виконує деякі операції з покажчиками порядків вище першого:

```
#include <iostream>
using namespace std;

int main()
{
    int var = 123; // ініціалізація змінної var чис-
лом 123
```

```
    int *ptrvar = &var; // покажчик на змінну var
    int **ptr_ptrvar = &ptrvar; // покажчик на по-
кажчик на змінну var
    int ***ptr_ptr_ptrvar = &ptr_ptrvar;
    cout << " var\t\t= " << var << endl;
    cout << " *ptrvar\t= " << *ptrvar << endl;
    cout << " **ptr_ptrvar   = " << **ptr_ptrvar <<
endl; // два рази розіменовуємо покажчик, через те,
що він другого порядку
    cout << " ***ptr_ptrvar  = " << ***ptr_ptr_
ptrvar << endl; // покажчик третього порядку
    cout << "\n ***ptr_ptr_ptrvar -> **ptr_ptrvar ->
*ptrvar ->      var -> "<< var << endl;
    cout << "\t  " << &ptr_ptr_ptrvar<< " -> " << "
" << &ptr_ptrvar << " ->" << &ptrvar << " -> " <<
&var << " -> " << var << endl;
    return 0;
}
```

Результат роботи програми наведений нижче:

```
var          = 123
 *ptrvar     = 123
 **ptr_ptrvar   = 123
 ***ptr_ptrvar  = 123

 ***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar ->
var -> 123
        0xffffcc20 ->     0xffffcc28 ->0xffffcc30 ->
0xffffcc3c -> 123
```

Дана програма доводить той факт, що для отримання значення, кількість розіменування покажчика має збігатися з його порядком. Логіка n-кратного розіменування полягає в тому, що програма послідовно перебирає адреси всіх покажчиків аж до змінної, у якій міститься значення. У програмі показана реалізація покажчика третього порядку. І якщо, використовуючи такий покажчик (третього порядку), необхідно отримати значення, на яке він посилається, робиться 4 кроки:

1. За значенням покажчика третього порядку отримати адресу покажчика другого порядку.

2. За значенням покажчика другого порядку отримати адресу покажчика першого порядку.

3. За значенням покажчика першого порядку отримати адресу змінної.

4. За адресою змінної отримати доступ до її значення.

Покажчики на функції

Покажчики можуть посилатися на функції. Ім'я функції, як і ім'я масиву само по собі є покажчиком, тобто містить адресу входу.

```
// оголошення покажчика на функцію
/* Тип даних */ (* /* ім'я покажчика */) (/* список аргументів функції */);
```

Тип даних визначаємо такий, який буде повертати функція, на яку буде посилатися покажчик. Символ покажчика і його ім'я беруться в круглі дужки, щоб показати, що це покажчик, а не функція, яка повертає покажчик на певний тип даних. Після імені покажчика йдуть круглі дужки, у цих дужках перераховуються всі аргументи через кому, як в оголошенні прототипу функції. Аргументи успадковуються від тієї функції, на яку буде посилатися покажчик.

Розглянемо програму, яка використовує покажчик на функцію. Програма знаходить НСД – найбільший спільний дільник. НСД – це найбільше ціле число, на яке без залишку діляться два числа, введених користувачем. Вхідні числа також повинні бути цілими.

```
#include <iostream>
using namespace std;

int gcd(int, int ); // прототип вказаної функції

int main()
{
```

```
    int (*ptrgcd)(int, int); // оголошення покажчика
на функцію
    ptrgcd=gcd; // присвоюємо адресу функції покаж-
чику ptrgcd
    int a, b;
    cout << "Enter first number: ";
    cin >> a;
    cout << "Enter second number: ";
    cin >> b;
    cout << "GCD = " << ptrgcd(a, b) << endl; //
звертаємось до функції через покажчик
    return 0;
}

int gcd(int number1, int number2) // рекурсивна функ-
ція знаходження найбільшого спільного дільника GCD
{
    if ( number2 == 0 ) // базове розв'язання
        return number1;
    return gcd(number2, number1 % number2); // ре-
курсивне розв'язання НОД
}
```

Результат роботи програми:

```
Enter first number: 1001
Enter second number: 65
GCD = 13
```

Динамічні масиви в C++

Динамічне виділення пам’яті необхідно для ефективного використання пам’яті комп’ютера. Наприклад, ми написали якусь програму, яка обробляє масив. Під час написання даної програми необхідно було оголосити масив, тобто задати йому фіксований розмір (наприклад, від 0 до 100 елементів), тоді ця програма буде не універсальною, адже може обробляти масив розміром не більше 100 елементів. А якщо нам знадобляться всього 20 елементів, але в пам’яті виділиться місце під 100 елементів, адже

оголошення масиву було статичним, а таке використання пам'яті вкрай неефективне.

У C++ операції `new` і `delete` призначені для динамічного розподілу пам'яті комп'ютера. Операція `new` виділяє пам'ять з області вільної пам'яті, а операція `delete` звільняє виділену пам'ять. Пам'ять, яка виділяється, після її використання повинна вивільнятися, тому операції `new` і `delete` використовуються парами. Навіть якщо не вивільняти пам'ять явно, то вона звільниться ресурсами ОС по завершенню роботи програми. Рекомендується все ж таки не забувати про операцію `delete`.

```
// приклад використання операції new
int * ptrvalue = new int;
// де ptrvalue - покажчик на виділену ділянку пам'яті типу int
// new - операція виділення вільної пам'яті під створюваний об'єкт.
```

Операція `new` створює об'єкт заданого типу, виділяє йому пам'ять і повертає покажчик правильного типу на дану ділянку пам'яті. Якщо пам'ять неможливо виділити, наприклад, у разі відсутності вільних ділянок, то повертається нульовий покажчик, тобто покажчик зі значенням 0 (nullptr). Виділення пам'яті можливо під будь-який тип даних: `int, float, double, char` і т. д.

```
// приклад використання операції delete:
delete ptrvalue;
// де ptrvalue - покажчик на виділену ділянку пам'яті типу int
// delete - операція вивільнення пам'яті
```

Розглянемо програму, у якій буде створюватися динамічна змінна:

```
#include <iostream>
using namespace std;
```

```
int main()
{
     int *ptrvalue = new int; // динамічне виділення
пам'яті під об'єкт типу int
    *ptrvalue = 9; // ініціалізація об'єкта через по-
кажчик
    //int *ptrvalue = new int (9); ініціалізація може
виконуватися одразу при оголошенні динамічного об'єк-
та
    cout << "ptrvalue = " << *ptrvalue << endl;
    delete ptrvalue; // вивільнення пам'яті
    return 0;
}
```

Створення динамічних масивів

Найчастіше операції `new` і `delete` застосовуються для створення динамічних масивів, а не для створення динамічних змінних. Розглянемо фрагмент коду створення одновимірного динамічного масиву.

```
// оголошення одновимірного динамічного масиву на 10
елементів:
float * ptrarray = new float [10];
// де ptrarray - покажчик на виділену ділянку пам'я-
ті під масив дійсних чисел типу float
// у квадратних дужках вказуємо розмір масиву
```

Після того як динамічний масив став непотрібним, необхідно звільнити ділянку пам’яті, яка під нього виділялась.

```
delete [] ptrarray;
```

Після оператора `delete` ставляться квадратні дужки, які говорять про те, що вивільняється ділянка пам’яті, що відводилась під одновимірний масив. Розглянемо програму, у якій створюється одновимірний динамічний масив, заповнений випадковими числами.

```
#include <iostream>
#include <ctime>
#include <iomanip>
```

```
using namespace std;

int main()
{
    srand(time(0)); // генерація випадкових чисел
    float *ptrarray = new float [10]; // створення ди-
намічного масиву дійсних чисел на десять елементів
    for (int count = 0; count < 10; count++)
        ptrarray[count] = (rand() % 10 + 1) /
float((rand() % 10 + 1)); // заповнення масиву випад-
ковими числами з масштабуванням від 1 до 10
    cout << "array = ";
    for (int count = 0; count < 10; count++)
        cout << setprecision(2) << ptrarray[count]
<< "    ";
    delete [] ptrarray; // вивільнення пам'яті
    cout << endl;
    return 0;
}
```

Створений одновимірний динамічний масив заповнюється випадковими числами, отриманими за допомогою функцій генерації випадкових чисел, причому числа генеруються в інтервалі від 1 до 10, інтервал задається так – `rand()% 10 + 1`. Щоб отримати випадкові дійсні числа, виконується операція ділення, з використанням явного приведення до дійсного типу знаменника – `floatrand ()% 10 + 1`. Щоб показати тільки два знаки після коми, використовуємо функцію `setprecision(2)`, прототип даної функції знаходиться в заголовку `<iomanip>`. Функція `time(0)` засіває генератор випадкових чисел тимчасовим значенням, таким чином, виходить, відтворювати випадковість виникнення чисел

```
array = 1.6    1    0.5    1    4    0.22    4.5
1.3    3    1.6
```

По завершенню роботи з масивом, він видаляється, таким чином, вивільняється пам’ять, відведена під його зберігання.

Тепер розглянемо фрагмент коду, у якому показано, як оголошується двовимірний динамічний масив.

```
// оголошення двовимірного динамічного масиву на 10
елементів:
float **ptrarray = new float* [2]; // два рядки в масиві
     for (int count = 0; count < 2; count ++)
         ptrarray [count] = new float[5]; // і п'ять
стовпців
// де ptrarray - масив покажчиків на виділену ділян-
ку пам'яті під масив дійсних чисел типу float
```

Спочатку оголошується покажчик другого порядку `float **ptrarray`, який посилається на масив покажчиків `float* [2]`, де розмір масиву дорівнює двом. Після чого в циклі for кожному рядку масиву оголошеного в рядку 2 виділяється пам'ять під п'ять елементів. У результаті виходить двовимірний динамічний масив `ptrarray[2][5]`. Розглянемо приклад вивільнення пам'яті, відводиться під двовимірний динамічний масив.

```
// вивільнення пам'яті, яка відводиться під двовимір-
ний динамічний масив:
     for (int count = 0; count < 2; count ++)
         delete [] ptrarray[count];
// де 2 - кількість рядків у масиві
```

Оголошення та видалення двовимірного динамічного масиву виконується за допомогою циклу, тому необхідно зрозуміти та запам'ятати те, як це робиться. Розглянемо програму, у якій створюється двовимірний динамічний масив.

```
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;

int main ()
{
    srand(time (0)); // генерація випадкових чисел
```

```
    // динамічне створення двовимірного масиву дійс-
них чисел на десять елементів
    float **ptrarray = new float* [2]; // два рядки
в масиві
    for (int count = 0; count < 2; count ++)
        ptrarray [count] = new float[5]; // і п'ять
стовпців
    // заповнення масиву
    for (int count_row = 0; count_row < 2; count_row
++)
        for (int count_column = 0; count_column < 5;
count_column ++)
            ptrarray [count_row] [count_column] =
(rand ()% 10 + 1) / float ((rand ()% 10 + 1)); // за-
повнення масиву випадковими числами з масштабуванням
від 1 до 10
    // вивід масиву
    for (int count_row = 0; count_row < 2; count_row
++)
    {
        for (int count_column = 0; count_column < 5;
count_column ++)
            cout << setw(5) << setprecision(2) <<
ptrarray [count_row] [count_column] << "";
        cout << endl;
    }
    // видалення двовимірного динамічного масиву
    for (int count = 0; count < 2; count ++)
        delete [] ptrarray[count];
    return 0;
}
```

Під час виведення масиву була використана функція `setw()`, вона відводить місце заданого розміру під виведені дані. У нашому випадку, під кожен елемент масиву по чотири позиції, це дозволяє вирівняти у стовпцях числа різної довжини.

```
    6    1  1.1    1    6
 0.62 0.25  1.3  1.1    1
```

Лабораторний практикум № 11

Мета: набути навичок створення та реалізації програм, що використовують динамічний розподіл пам'яті.

Завдання

Завдання 11.1 (C++)

Створити структуру, що була визначена в попередній лабораторній роботі. Визначити функції, що дозволяють створювати змінну типу створеної структури та вводити дані з клавіатури. Реалізувати зберігання даних в оперативній пам'яті у вигляді динамічного масиву та на диску – у файлі.

Програма повинна під час старту перевірити наявність файлу з даними на диску, якщо файл є – прочитати його вміст у динамічний масив, якщо файл відсутній – створити його.

Під час завершення роботи програма повинна зберегти всі дані, розміщені в динамічному масиві структур у файл.

Програма має містити функцію меню, що дозволяє користувачеві обирати потрібну йому дію: додавання запису, вилучення запису за вказаним id, виведення всіх даних, що зберігаються в масиві, у табличній формі, виведення даних, відповідно до запитів (параметри запитів вводити з клавіатури).

1. **Student:** id, Прізвище, Ім'я, По батькові, Дата народження, Адреса, Телефон, Факультет, Курс, Група.

Запити:

a. Список студентів указаного факультету.

b. Список студентів, які народилися після вказаного року.

c. Список студентів, чиї номери телефонів починаються із вказаної послідовності цифр.

d. Список навчальної групи в алфавітному порядку.

2. **Customer:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Номер банківського рахунку.

Запити:

a. Список покупців в алфавітному порядку.

b. Список покупців, у яких номер кредитної картки знаходиться в заданому інтервалі.

c. Список покупців, у яких адреса включає в себе вказану послідовність літер (наприклад, назву міста).

d. Список покупців, у яких номер банківського рахунку закінчується на вказану цифру.

3. **Patient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Номер медичної картки, Діагноз.

Запити:

a. Список пацієнтів, які мають указаний діагноз.

b. Список пацієнтів, чий номер медичної картки містить указану послідовність цифр.

c. Список пацієнтів, у яких адреса починається із вказаної послідовності символів.

d. Список пацієнтів, номер медичної карти яких знаходиться в заданому інтервалі.

4. **Abiturient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Оцінки.

Запити:

a. Список абітурієнтів, які мають незадовільні оцінки.

b. Список абітурієнтів, у яких сума балів вище заданої.

c. Список абітурієнтів, у яких номер телефону починається із заданої послідовності цифр (інші символи номера ігноруються).

d. Вибрати вказану кількість n абітурієнтів, які мають найбільшу суму балів.

5. **Book:** id, Назва, Автор(и), Видавництво, Рік видання, Кількість сторінок, Ціна, Тип палітурки.

Запити:

a. Список книг заданого автора.

b. Список книг, що видані вказаним видавництвом.

c. Список книг, кількість сторінок у яких належить указаному діапазону.

d. Список книг, що видані після заданого року.

6. **House:** id, Номер квартири, Площа, Поверх, Кількість кімнат, Вулиця, Тип будівлі, Термін експлуатації.

Запити:

a. Список квартир, що мають задану кількість кімнат.

b. Список квартир, що мають указану кількість кімнат і розташованих між указаними поверхами.

c. Список квартир, які експлуатуються не більше R (ввести з клавіатури) років, що знаходяться на вказаній вулиці.

d. Список квартир, які мають площу, що більше заданої.

7. **Phone:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Час міських розмов, Час міжнародних розмов.

Запити:

a. Відомості про абонентів, у яких час міських розмов перевищує вказаний.

b. Відомості про абонентів, які користувалися міжнародним зв'язком.

c. Відомості про абонентів, номер кредитної картки яких закінчується на вказану послідовність цифр.

d. Відомості про абонентів в алфавітному порядку.

8. **Car:** id, Марка, Модель, Рік випуску, Колір, Ціна, Реєстраційний номер.

Запити:

a. Список автомобілів заданої марки.

b. Список автомобілів заданої моделі, які експлуатуються більше n років.

c. Список автомобілів указаного кольору, реєстраційний номер яких містить указану послідовність цифр.

d. Список автомобілів указаного року випуску, ціна яких більше вказаної.

9. **Product:** id, Найменування, Тип, Виробник, Ціна, Термін зберігання, Кількість.

Запити:

a. Список товарів заданого найменування.

b. Список товарів заданого найменування, ціна яких не більше заданої.

c. Список товарів указаного типу заданого виробника.

d. Список товарів, термін зберігання яких більше заданого.

10. **Train:** id, Пункт призначення, Номер поїзда, Час відправлення, Число місць (загальних, плацкарт, купе, люкс).

Запити:

a. Список поїздів, що прямують до заданого пункту призначення.

b. Список поїздів, що прямують до заданого пункту призначення та відправляються після вказаної години.

c. Список поїздів, у яких кількість плацкартних місць більше ніж усіх інших разом.

d. Список поїздів, що відправляються до заданого пункту призначення та мають загальні місця.

Завдання 11.2 (Kotlin)

Виконати завдання 11.1, застосувавши мову програмування Kotlin. Замість структур (C++) використовувати Data Classes. Замість динамічних масивів використовувати списки.

Контрольні питання

1. Як відбувається розподіл пам’яті в C++?
2. Чому потрібно звільняти пам’ять, що не використовується?
3. Як розподіляється пам’ять для динамічних масивів?
4. Що таке покажчики вищих порядків?
5. Як відбувається отримання адреси змінної та розіменування покажчиків?

Тема 12. ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ. ЛІНІЙНІ ЗВ'ЯЗАНІ СПИСКИ

Основні поняття

Лінійні списки – найпростіша форма організації даних з рекурсивною структурою. Цю форму застосовують найчастіше тоді, коли треба обробити деяку сукупність даних, кількість елементів яких заздалегідь не визначена, а взаємний порядок проходження відіграє важливу або вирішальну роль. Лінійний односпрямований (однозв'язний) список – дані динамічної структури, що становлять собою сукупність лінійно зв'язаних однорідних елементів, до яких дозволяється додавати елементи до голови (початку) або в кінець (хвіст) списку, між будь-якими двома іншими, та видаляти будь-який елемент [7].

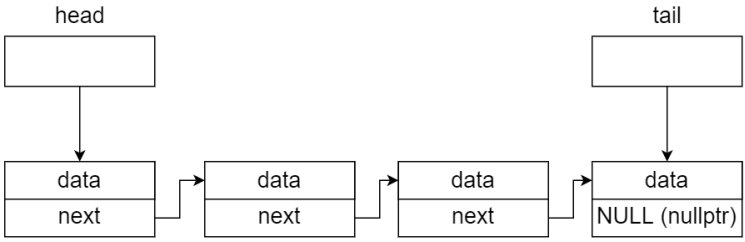


Рис. 12.1. Лінійний динамічний список

```
// C++
struct element
{
```

```
    typeelem data;
    element *next;
};

// Kotlin
data class Element<T>(var data: T, var next: Element<T>?
= null)
```

Також під час реалізації лінійного зв'язаного списка мовою Kotlin, необхідно створити клас List, що буде мати посилання head та tail, і всі операції реалізовуватимуться як функції в цьому класі.

```
// Kotlin
class List<T> {
    var head : Element<T>? = null
    var tail : Element<T>? = null
// ...
}
```

Для роботи з лінійним однозв'язним списком потрібні такі покажчики:

1. На голову списку Head.
2. На кінець списку Tail, але можна обійтися і без нього.
3. На k-й елемент списку.
4. Тимчасовий для виділення пам'яті під елементи, які додаються, і для звільнення елементів, що видаляються (ідентифікатор p).

Оскільки в мові Kotlin немає покажчиків, використовуються дуже схожі на них за сенсом, але більш безпечні **посилання**.

Розглянемо основні дії над однозв'язним списком.

Примітка. Тут і далі, під терміном «покажчик» ми будемо розуміти покажчики (указівники) у мовах C/C++ та посилання в мові Kotlin.

Ініціалізація списку

У разі застосування покажчика на кінець списку ініціалізація буде мати такий вигляд:

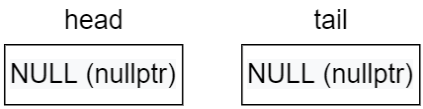


Рис. 12.2. Ініціалізація списку

```
// C++
element *head, *tail;
head = nullptr;
tail = nullptr;
або коротшим записом:
element *head = nullptr, *tail = nullptr;

// Kotlin
var head : Element<T>? = null
var tail : Element<T>? = null
```

Якщо покажчик на кінець списку не використовують, ініціалізувати треба тільки один покажчик:

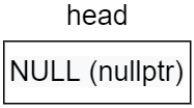


Рис. 12.3. Ініціалізація списку з одним покажчиком

```
// C++
element *head;
head = nullptr;
// або одним рядком:
element *head = nullptr;

// Kotlin
var head : Element<T>? = null
```

Таким чином, список ініціалізований, але не містить жодного елемента. Тепер можна заповнювати список з голови (додаючи знову створений елемент списку після попередніх) або з кінця (додаючи знову створений елемент перед попереднім).

Додавання елементів

Додавання елементів до списку з використанням двох покажчиків (на голову і кінець списку)

Додавання першого елемента до списку та додавання елемента в кінець списку відрізняється. У будь-якому випадку необхідний ще один покажчик p на поточний (новий) елемент.

Додавання першого елемента до списку

1. Вихідний стан:

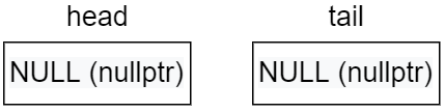


Рис. 12.4. Додавання елементів (вихідний стан)

```
// C++
element *head = nullptr, *tail = nullptr;
element *p;
// Kotlin
var head: Element<T>? = null
var tail: Element<T>? = null
var p: Element<T>? = null
```

2. Виділення пам’яті під перший елемент та занесення інформації до нього:

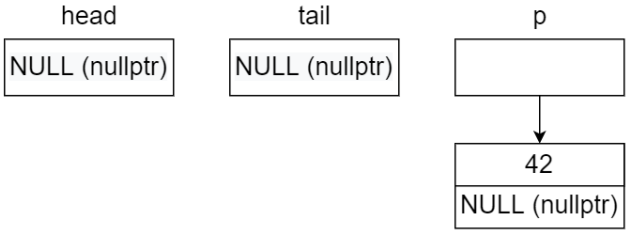


Рис. 12.5. Виділення пам’яті під перший елемент

```
// C++
p = new element;   // виділення пам´яті під новий елемент
```

```
p->data = 42;        // присвоєння значення інформа-
ційній частині
p->next = nullptr;   // присвоєння покажчику на на-
ступний
                     // елемент значення ознаки кінця

// Kotlin
p = Element(42)      // присвоєння значення інформа-
ційній частині,
                     // посилання на наступний еле-
мент автоматично
                     // отримає значення null
```

Опис покажчика на поточний елемент і виділення пам'яті краще записувати одним рядком (у мові Kotlin було саме так):

```
// C++
element *p = new element;
```

3. Установлення покажчиків head, tail на створений перший елемент:

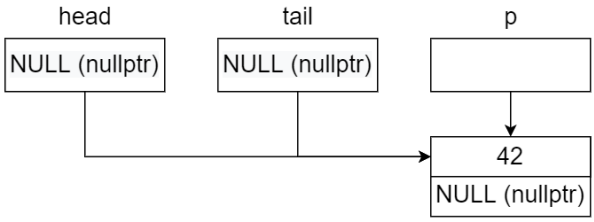


Рис. 12.6. Установлення покажчиків списку на перший елемент

```
// C++
head = p;
tail = p;
// Kotlin
head = p
tail = p
```

Додавання елемента в кінець списку

1. Вихідний стан:

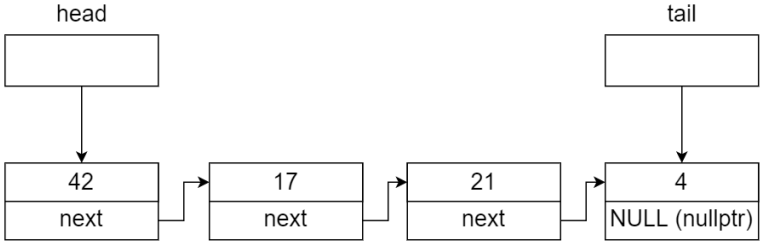


Рис. 12.7. Додавання елемента в кінець списку. Початковий стан

2. Виділення пам’яті під новий елемент списку й занесення інформації до нього:

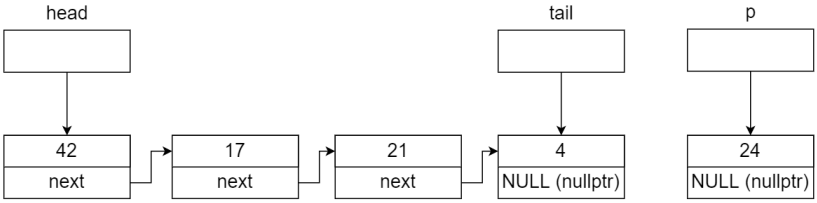


Рис. 12.8. Виділення пам’яті для елемента

```
// C++
element *p = new element;
p->data = 24;         // присвоєння значення інформа-
ційній частині
p->next = nullptr;    // присвоєння покажчику на на-
ступний елемент значення ознаки кінця

// Kotlin
val p = Element(24)
```

3. Установлення зв’язку між останнім елементом списку й новим, а також переміщення покажчика кінця списку на новий елемент:

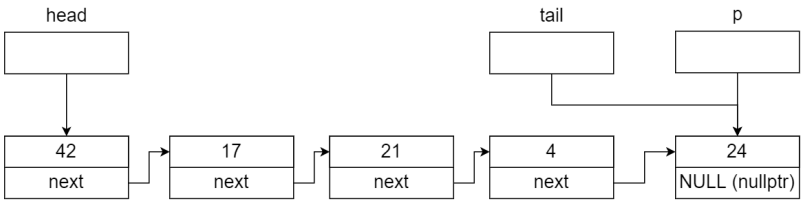


Рис. 12.9. Установлення зв'язку між останнім елементом та новим

```
// C++
tail->next = p;
tail = p;

// Kotlin
tail!!.next = p
tail = p
```

Таким чином, повністю функція додавання елемента в кінець списку з використанням покажчиків head, tail мовою C++ може мати такий вигляд:

```
void addToList(element **head, element **tail,
typeelem value)
{
    // виділення пам'яті під новий елемент списку
    element *p = new element;
    // заповнення інформаційної частини
    p->data = value;
    // встановлення покажчика останнього елемента
    p->next = nullptr;
    //якщо список порожній
    if (*head == nullptr) // або if (!(*head))
        // встановлення покажчика head на перший
елемент
        *head = p;
        // інакше встановлення зв'язку між останнім
елементом
        // списку й новим
    else (*tail)->next = p;
```

```
    // встановлення покажчика кінця списку на новий
елемент
    *tail = p;
}
```

Та ж сама операція мовою Kotlin:

```
fun addToList(value: T) {
    // виділення пам'яті під новий елемент списку,
заповнення інформаційної частини та встановлення по-
силання останнього елемента
    val p = Element(value)
    // якщо список порожній
    if (head==null) {
        // встановлення посилання head на перший
елемент
        head = p
    } else {
        // інакше встановлення зв'язку між останнім
елементом
        // списку й новим
        tail!!.next = p
    }
    // встановлення посилання кінця списку на новий
елемент
    tail = p
}
```

У реалізації мовою C++ під час виклику треба вказувати адреси покажчиків (покажчик на покажчик) head і tail.

```
// C++
addToList(&head, &tail, value)

// Kotlin дещо простіше:
list.addToList(value)
```

Додавання елементів у список з використанням одного покажчика (на голову списку)

Додавання першого елемента до списку майже не відрізняється від аналогічної операції з використанням двох покажчиків.

Додавання першого елемента до списку

1. Вихідний стан:

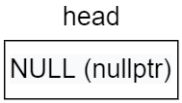


Рис. 12.10. Додавання елемента у список з одним покажчиком. Вихідний стан

```
// C++
head = nullptr;

// Kotlin
head = null
```

2. Виділення пам'яті під перший елемент списку й занесення інформації до нього:

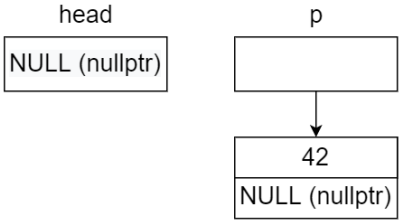


Рис. 12.11. Виділення пам'яті під перший елемент

```
// C++
element *p= new element;
p->data = 42;
p->next = nullptr;

// Kotlin
val p = Element(42)
```

3. Установлення покажчика head на створений перший елемент:

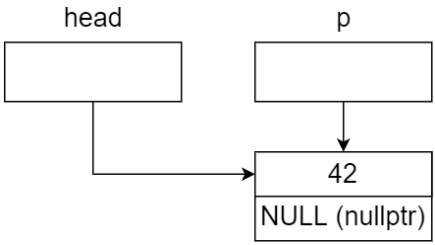


Рис. 12.12. Установлення покажчика на створений перший елемент

Ця дія записується мовами C++ та Kotlin майже однаково:

```
// C++
head = p;

// Kotlin
head = p
```

Додавати елементи можна й до вже існуючого списку: у голову списку, усередину, після заданого та перед заданим елементом.

Додавання елемента в голову списку

1. Вихідний стан:

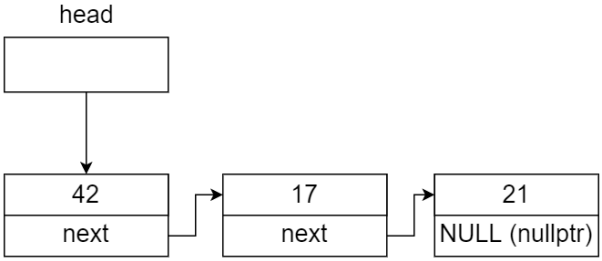


Рис. 12.13. Додавання елемента в голову списку. Вихідний стан

2. Виділення пам’яті під новий елемент списку й заповнення інформаційного поля:

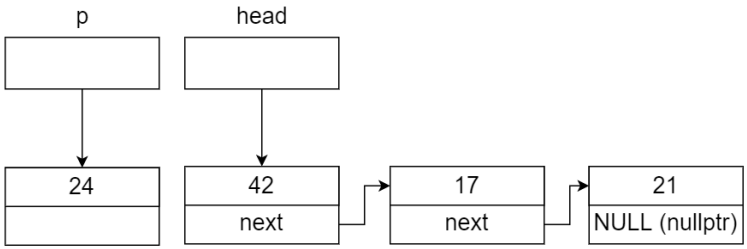


Рис. 12.14. Виділення пам'яті під новий елемент

```
// C++
element *p = new element;
p->data = 24;

// Kotlin
val p = Element(24)
```

3. Установлення зв'язку між першим елементом списку й новим:

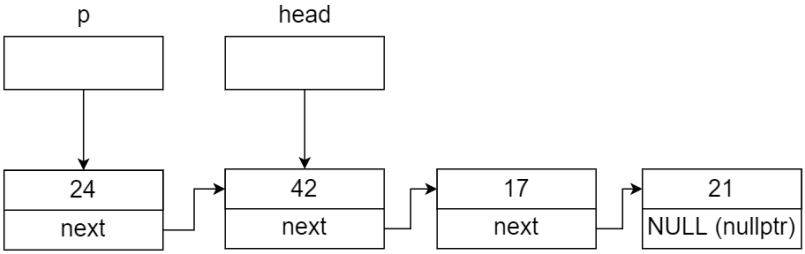


Рис. 12.15. Установлення зв'язку
між першим елементом списку та новим

```
// C++
p->next = head;

// Kotlin
p.next = head
```

4. Переміщення покажчика на голову списку на новий елемент:

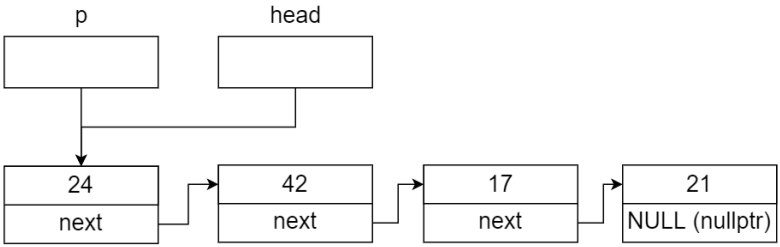


Рис. 12.16. Переміщення покажчика
на голову списку на новий елемент

```
// C++
head = p;

// Kotlin
head = p
```

Таким чином, остаточно функція додавання елемента в голову списку може мати такий вигляд:

Мовою C++

```
void addInHead(element **head, typeelem value){
    // виділення пам'яті під новий елемент списку
    element *p = new element;
    // заповнення інформаційної частини
    p->data = value;
    // встановлення зв'язку між першим елементом
списку й новим
    p->next = *head;
    // Переміщення покажчика на голову на новий еле-
мент
    *head = p;
}
```

Мовою Kotlin

```
fun addInHead(value: T) {
    // виділення пам'яті під новий елемент списку
    // та заповнення інформаційної частини
    val p = Element(value)
    // встановлення зв'язку між першим елементом
списку й новим
```

```
    p.next = head
    // переміщення посилання на голову на новий еле-
мент
    head = p
}
```

Виведення елементів списку, починаючи від голови

Для виведення елементів списку на екран потрібно використовувати допоміжне посилання, якому на початку надається значення голови списку. Після опрацювання кожного елемента (виведення на екран його інформаційної частини), відбувається перехід до наступного. Процес продовжується доки покажчик не стане дорівнювати `nullptr` через досягнення кінця списку:

```
// C++
void printList(element *head) {
    element *p = head;
    while (p!=nullptr) {
        cout << p->data << " ";
        p = p->next;
    }
}

// Kotlin
fun printList() {
    var p = head
    while (p!=null) {
        println("${p.data} ")
        p = p.next
    }
}
```

Також можна написати рекурсивну функцію виведення елементів списку:

```
// C++
void printListRec(element *head) {
    if (head != nullptr) {
        cout << head->data << " ";
```

```
        printListRec(head->next);
    }
}

// Kotlin
fun printListRec(head: Element<T>?) {
    if (head != null) {
        print("${head.data} ");
        printListRec(head.next)
    }
}
```

Для виведення всього списку надані функції треба викликати з фактичним параметром head, який зберігає адресу першого елемента. Функції придатні також і для виведення на екран частини списку, для цього як фактичний параметр треба передати адресу елемента, з якого буде починатися виведення списку.

Пошук елемента з певними властивостями

Функція пошуку елемента у списку, текст якої наведено нижче, повертає покажчик на той елемент списку, що містить у своїй інформаційній частині значення, задане користувачем; якщо ж такий елемент не знайдено, функція повертає **nullptr**.

Функції передають два параметри: покажчик на голову списку, у якому буде відбуватися пошук і значення інформаційної частини елемента списку, яке необхідно знайти. Алгоритм пошуку дуже простий: будемо послідовно переглядати елементи списку й порівнювати значення інформаційного поля із заданим значенням. Цей процес закінчується у двох випадках:

- черговий елемент списку містить задане значення, тоді функція повертає покажчик на даний елемент та припиняє свою роботу;

- список було вичерпано, тобто повністю переглянуто, але задане значення не знайдено; тоді функція повертає «порожнє» посилання `nullptr` (`null` – у мові Kotlin).

```
// C++
element * findNode(element * head, typeelem x)
{
    // покажчик на перший елемент списку
    element * node = head;
    while(node != nullptr) {    // або while(node)
        //якщо заданий елемент знайдено
        if (node->data == x)
       //закінчення пошуку і повернення покажчика на
цей елемент
            return node;
            // у іншому випадку
        else
            // перехід на наступний елемент списку
            node = node->next;
    }
    // якщо список вичерпано, то шуканий елемент не
знайдено,
    // тому повертаємо "порожнє" значення
    return nullptr;
}

// Kotlin
fun findNode(head : Element<T>?, x: T) : Element<T>? {
    var p = head
    while (p != null) {
        if (p.data == x) break
        p = p.next
    }
    return p
}
```

Додавання елемента всередину списку *після* заданого елемента

Вважаємо, що адреса заданого елемента відома і зберігається у покажчику `pk`. Для додавання нового елемента після заданого необхідно:

1. Створити новий динамічний об'єкт (новий елемент списку).

2. У поле `data` об'єкта занести задану інформаційну частину.

3. У поле `next` даного об'єкта занести посилання, взяте з відповідного поля того елемента, за яким повинен іти новий елемент (покажчик `pk`).

4. У поле `next` того елемента, за яким повинен іти новий елемент, занести посилання на цей елемент (покажчик `pk`).

1. Вихідний стан:

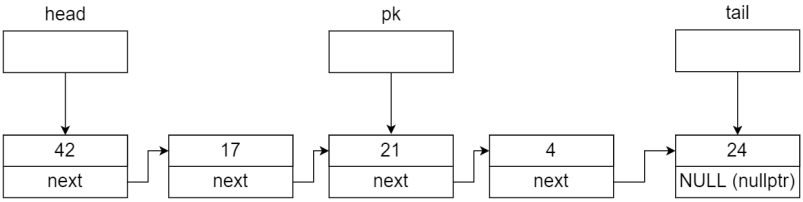


Рис. 12.17. Додавання елемента після заданого елемента. Вихідний стан

2. Виділення пам'яті під новий елемент списку й заповнення інформаційного поля:

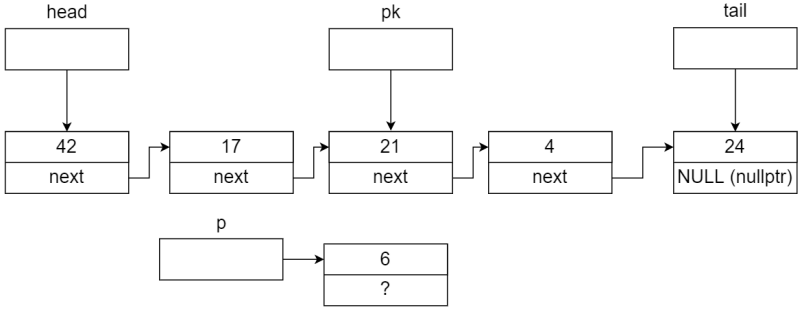


Рис. 12.18. Виділення пам'яті та заповнення інформаційного поля

```
// C++
element * p= new element;
p->data = 6;

// Мовою Kotlin
val p = Element(6)
```

3. Установлення зв'язку між новим і наступним за ним елементом:

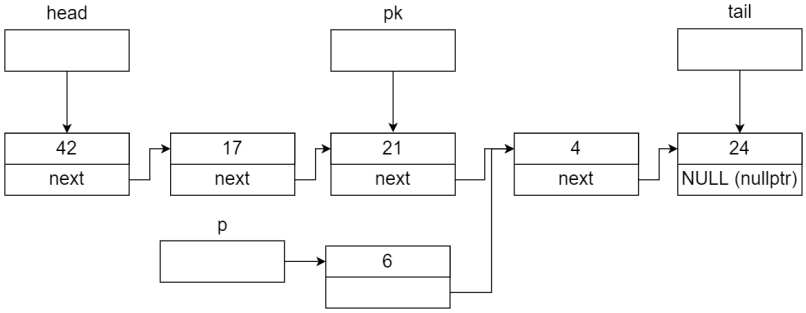


Рис. 12.19. Установлення зв'язку
між новим і наступним за ним елементом

```
// C++
p->next = pk->next;

// Kotlin
p.next = pk.next
```

4. Перестановка зв'язку заданого елемента на новий елемент:

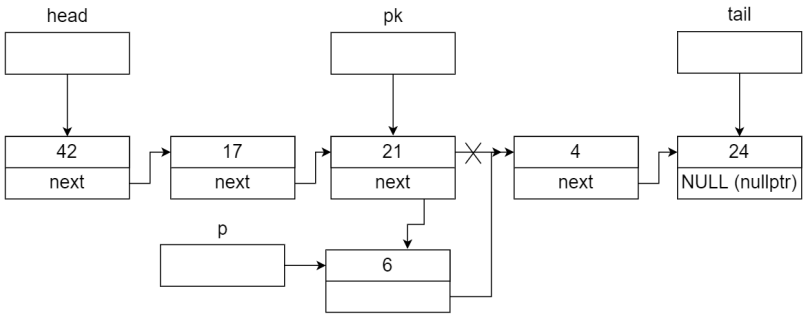


Рис. 12.20. Перестановка зв'язків

```
// C++
pk->next = p;

// Kotlin
pk.next = p
```

Таким чином, остаточно опис функції додавання у список заданого елемента після визначеного може мати такий вигляд:

Мовою C++

```
void addNodeAfter(element ** pk, typeelem value)
{
    //створення нового динамічного об'єкта
    element * p = new element;
    //запис інформаційної частини
    p->data = value;
    //заповнення покажчика на наступний елемент
    p->next = (*pk)->next;
    //додавання нового елемента у список
    (*pk)->next = p;
}
```

Мовою Kotlin

```
fun addNodeAfter(pk : Element<T>, value : T) {
    val p = Element(value)
    p.next = pk.next
    pk.next = p
}
```

Додавання елемента всередину списку *перед* заданим елементом

Якщо є адреса елемента, який передує заданому, то необхідну дію можна виконати так, як описано вище. Але якщо є адреса тільки заданого елемента, то задача ускладнюється.

У цьому випадку замість того, щоб ще раз від початку списку шукати попередній елемент, значно простіше виконати вставку перед заданим елементом у такий спосіб (вважаємо, що адреса заданого елемента відома і зберігається у вказівнику `pk`):

1. Зробити вставку нового елемента після заданого елемента (таким чином, як це описано вище).

2. Поміняти місцями значення інформаційних полів заданого й нового елементів.

3. Переставити покажчик `pk` на новий уставлений елемент, який вже містить значення заданого елемента.

1. Вихідний стан:

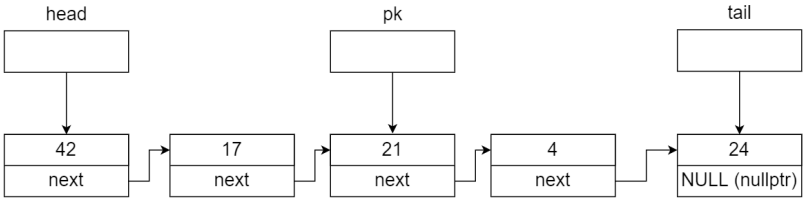


Рис. 12.21. Додавання елемента перед указаним. Вихідний стан

2. Стан після додавання елемента 6 після елемента 21.

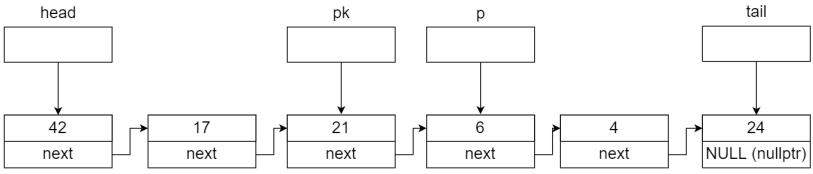


Рис. 12.22. Додавання елемента. Перший крок

```
// C++
element * p= new element;
p->data = 6;
p->next = pk->next;
pk->next = p;
// Kotlin
val p = Element(6)
p.next = pk.next
pk.next = p
```

3. Обмін місцями значень інформаційних полів заданого й нового елементів (змінна tmp повинна мати той же тип, що й інформаційне поле елемента списку):

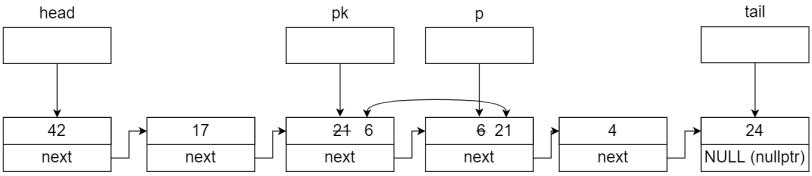


Рис. 12.23. Обмін місцями значень інформаційних полів елементів

```
// C++
typeelem tmp = pk->data;
pk->data = p->data;
p->data = tmp;
```

```
// Kotlin
val tmp = pk.data
pk.data = p.data
p.data = tmp
```

4. Перестановка покажчика pk на новий елемент:

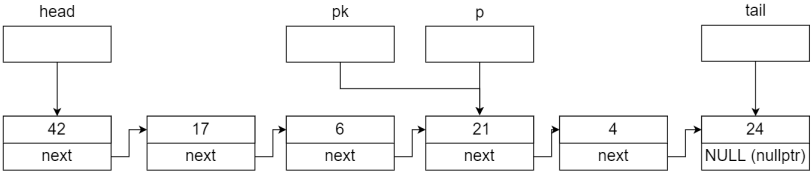


Рис. 12.24. Перестановка покажчика на новий елемент

```
// C++
pk = p;

// Kotlin
pk = p
```

Видалення елементів списку

Під час видалення першого та останнього елемента списку необхідно не загубити значення покажчика на голову та ознаку кінця списку.

Видалення першого елемента

1. Вихідний стан:

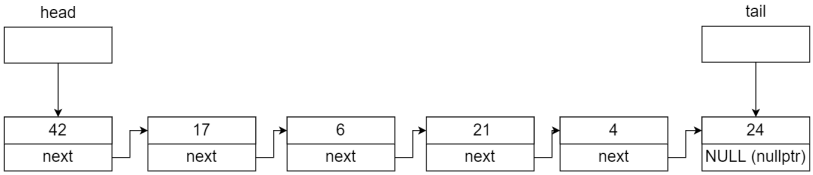


Рис. 12.25. Видалення елемента. Вихідний стан

2. Установлення додаткового покажчика p на елемент, який видаляють, і вибирання з нього інформації:

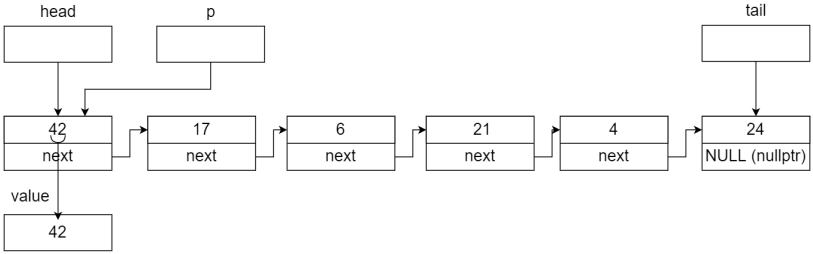


Рис. 12.26. Видалення елемента. Додатковий покажчик

```
// C++
element * p = head;
value = p->data;
```

```
// Kotlin
val value = head?.data  // додатковий покажчик не по-
трібний
```

3. Перестановка покажчика на голову списку на наступний елемент, звільнення пам'яті першого елемента списку:

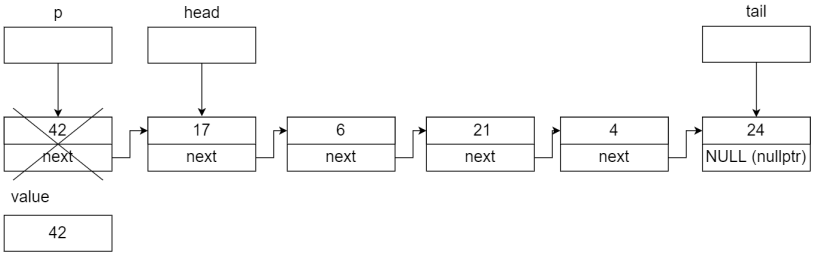


Рис. 12.27. Перестановка покажчика голови списку на новий елемент

```
// C++
head = head->next; // або head = p->next;
delete p;

// Kotlin
head = head.next
```

Остаточно функцію, що реалізує видалення першого елемента списку, наведено нижче.

Мовою C++

```
typeelem deleteFirst(element ** head)
{
    //зберігаємо адресу елемента, який потрібно ви-
далити
    element * p = *head;
    //отримуємо з нього інформацію
    typeelem value = p->data;
    //встановлюємо голову списку на наступний елемент
    *head = p->next;
    //видаляємо перший елемент
    delete p;
```

```
    // повертаємо значення видаленого елемента
    return value;
}
```

Мовою Kotlin

```
fun deleteFirst() : T? {
    val value = head?.data
    head = head?.next
    return value
}
```

Примітка. Зверніть увагу на те, що Kotlin не вимагає явного звільнення пам'яті. З іншого боку, Kotlin вимагає перевірки на null під час отримання значення за посиланням.

Видалення останнього елемента списку

1. Для видалення останнього елемента списку необхідно знати адресу передостаннього елемента для збереження у його посилальній частині ознаки кінця списку nullptr.

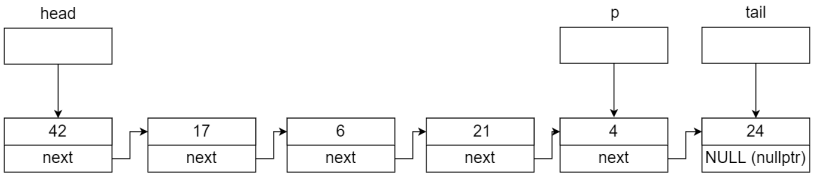


Рис. 12.28. Перестановка покажчика голови списку на новий елемент

```
// C++
element * p = head;
while (p->next->next != nullptr) p = p->next;

// Kotlin
val p = head
while (p?.next?.next != null) p = p.next
```

2. Якщо в реалізації списку використовується покажчик на останній елемент tail, вибираємо значення з нього, звільнюємо пам'ять, на яку він вказує. Якщо ж такого

елемента немає, то визначаємо його як temp = p→next; вибираємо значення з нього, звільнюємо пам'ять, на яку він указує.

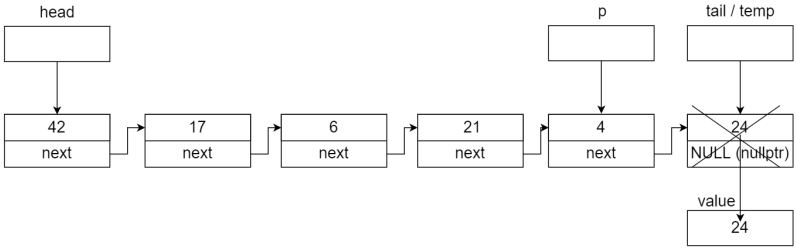


Рис. 12.29. Вибирання даних.
Вивільнення пам'яті останнього елемента

```
// C++
typeelement value = p->next->data;
// або value = tmp->data;
// або, якщо є покажчик tail: value = tail->data;
delete p->next;

// Kotlin
val value = p?.next?.data   // або val value =
tail?.data
```

3. Фіксація кінця списку (установлення посилального поля останнього елемента в nullptr та покажчика tail):

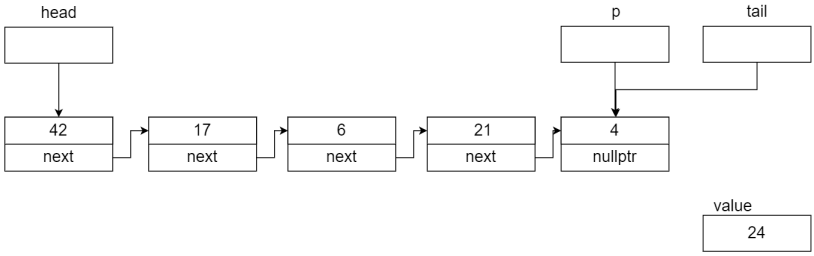


Рис. 12.30. Фіксація кінця списку

```
// C++
p->next = nullptr;
tail = p; // якщо покажчик tail використовується

// Kotlin
p?.next = null
tail = p
```

Отже, остаточно функцію, що реалізує видалення останнього елемента списку, описано нижче.

Мовою C++

```
typeelem deleteLast(element * head)
{
     //знаходження передостаннього елемента списку
     element * p = head;
     while (p->next->next != nullptr) p = p->next;
     //збереження адреси останнього елемента
     element * temp = p->next; // tail
     //вибирання з нього інформації
     typeelem value = p->next->data; // або value =
temp->data;
     //видалення останнього елемента
     delete temp;
     //збереження ознаки кінця списку
     p->next = nullptr;
     return value;
}
```

Мовою Kotlin

```
fun deleteLast() : T? {
    var p = head
    while (p?.next?.next != null) p = p.next
    val value = p?.next?.data
    p?.next = null
    tail = p
    return value
}
```

Видалення елемента, що стоїть *після* заданого

Для видалення елемента зі списку достатньо змінити посилання попереднього йому елемента, причому як нове

посилання цього елемента треба прийняти посилання елемента, який видаляємо. Варто звернути увагу на те, що в результаті виконання даної операції виключений зі списку елемент продовжує існувати й займати місце в пам'яті комп'ютера, хоча і стає недоступним для використання. Як бачимо, такий спосіб може призвести до неефективного використання пам'яті через зберігання в ній виключених елементів списку. Для усунення цього недоліку в описі функції видалення потрібно обов'язково передбачити знищення виключеного зі списку елемента:

1. Вихідний стан:

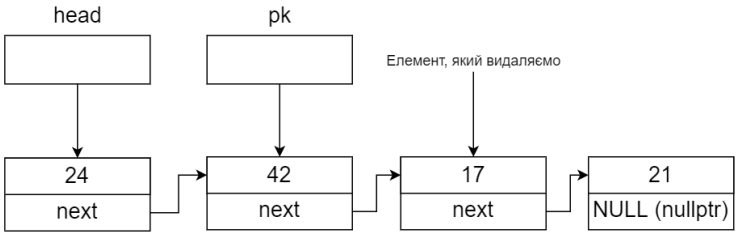


Рис. 12.31. Видалення елемента. Вихідний стан

2. Установлення додаткового покажчика p на елемент списку, який видаляємо, і вибирання з нього інформації:

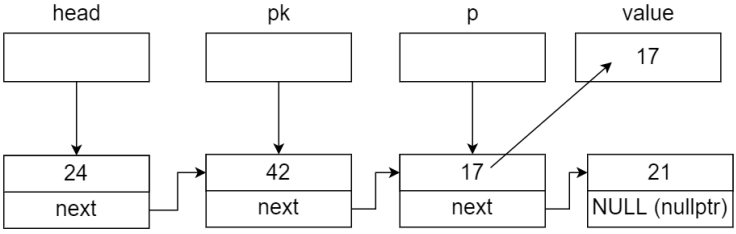


Рис. 12.32. Установлення додаткового посилання та вибирання інформації

```
// C++
element * p = pk->next;
```

```
typeelem value = p->data;

// Kotlin
val p = pk.next
val value = p?.data
```

3. Установлення зв'язку між k-м і (k+2)-м елементами та звільнення пам'яті (k+1)-го елемента, який видаляють:

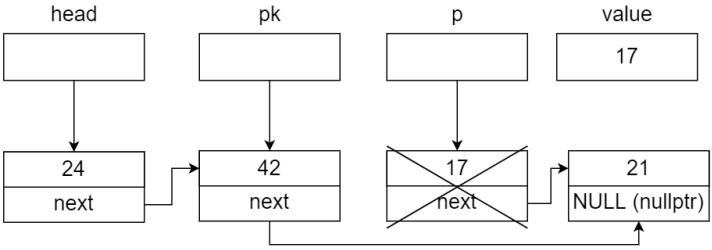


Рис. 12.33. Установлення нового зв'язку та звільнення пам'яті

```
// C++
pk->next = p->next;
//або
//pk->next = pk->next->next;
delete p;

// Kotlin
pk.next = p?.next
// або pk.next = pk.next?.next
```

Таким чином, функцію, що реалізує видалення елемента, який стоїть після заданого, можна описати так:

Мовою C++

```
typeelem deleteElementAfter(element * pk)
{
    // збереження посилання на елемент, який видаляємо
    element * p = pk->next;
    // збереження інформації
    typeelem value = pk->next;
    //змінюємо посилання, виключаючи елемент зі списку
```

```
    pk->next = pk->next->next;
    //або pk->next = p->next;
    //звільняємо пам'ять
    delete p;
    return value;
}
```

Мовою Kotlin

```
fun deleteElementAfter(pk : Element) : T? {
    val p = pk.next
    val value = p?.data
    pk.next = pk.next?.next
    return value
}
```

Видалення заданого елемента списку

Видалення заданого елемента можна здійснити також, як описано вище, якщо відома адреса попереднього елемента списку. Для цього можна застосувати функцію пошуку попереднього елемента findPrev:

```
// C++
element * findPrev(element * head, element * target)
{
    //поки не знайдено шуканий елемент
    while(head->next != target)
    {
        //перевіряємо, якщо список вичерпаний,
        if(!head)
            //повертаємо порожнє значення
            return nullptr;
        //або переходимо на наступний елемент
        head = head->Link;
    }
    //повертаємо шукану адресу
    return head;
}

// Kotlin
fun findPrev(head : Element<T>, target : Element<T>)
```

```
: Element<T> {
    var p = head
    while (p.next!=target) p = p.next!!
    return p
}
```

Однак можна обійтися і без додаткового пошуку, для чого потрібно:

1. Скопіювати значення інформаційного поля «потрібного» наступного елемента в інформаційне поле заданого елемента, який видаляємо.
2. Видалити наступний елемент замість заданого.

1. Вихідний стан:

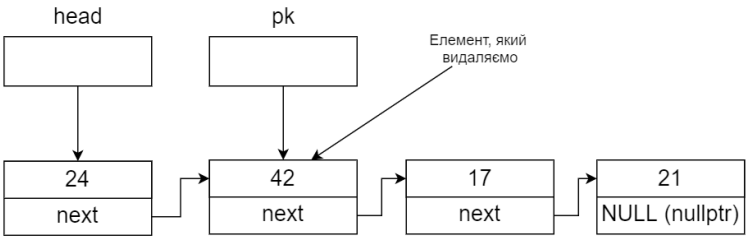


Рис. 12.34. Видалення елемента. Вихідний стан

2. Установлення додаткового покажчика p на наступний елемент списку, вибирання інформації із заданого елемента, який видаляємо, і копіювання корисної інформації з наступного елемента в заданий елемент:

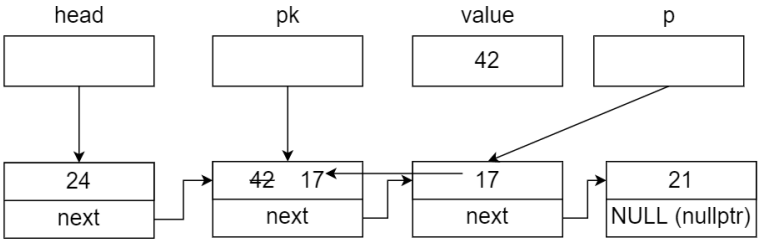


Рис. 12.35. Видалення елемента. Копіювання даних

```
// C++
element * p = pk->next;
typeelem value = pk->data;
pk->data = p->data;

// Kotlin
val p = pk.next
val value = pk.data
pk.data = p!!.data
```

3. Установлення зв'язку між k-м і (k+2)-м елементами і звільнення пам'яті (k+1)-го елемента, який видаляємо замість k-го:

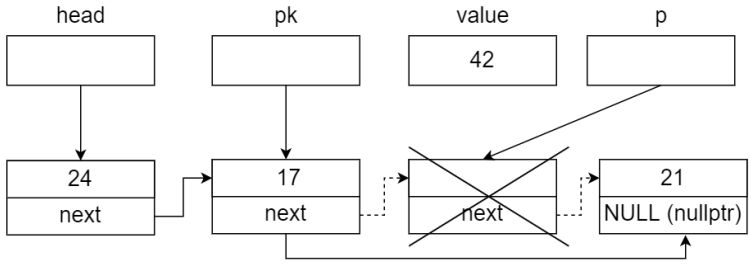


Рис. 12.36. Установлення зв'язку та звільнення пам'яті

```
// C++
pk->next = p->next;
//або
//pk->next = pk->next->next;
delete p;

// Kotlin
pk.next = p.next
```

Остаточно функцію, що реалізує видалення заданого елемента, можна подати таким чином:

Мовою C++

```
typeelem delElement(element * pk)
{
```

```
    // збереження посилання на елемент, який видаляємо
    element * p = pk->next;
    // збереження інформації
    typeelem value = pk->data;
    //обмін елементів місцями
    pk->data = p->data;
    //змінюємо посилання, виключаючи елемент зі
списку
    pk->next = pk->next->next;
    //або pk->next = p->next;
    //звільняємо пам'ять
    delete p;
    return value;
}
```

Мовою Kotlin

```
fun deleteElement(pk : Element<T>): T {
    val p = pk.next
    val value = pk.data
    pk.data = p!!.data
    pk.next = p.next
    return value
}
```

Видалення списку

Наприкінці роботи програми з однозв'язним списком необхідно видалити список повністю, щоб звільнити пам'ять, у протилежному разі після певної кількості запусків програми, що працює зі списками, можливі помилки у зв'язку з недоступністю комірок пам'яті через некоректне завершення роботи програми. Видалення кожного елемента списку аналогічне до видалення першого елемента списку:

• Зберігаємо адресу першого елемента в допоміжному покажчику.

• Покажчик на голову списку переміщуємо на другий елемент.

• Видаляємо перший елемент.

• Повторюємо ці дії, поки список не стане порожнім.

Або:

• Зберігаємо адресу другого елемента в допоміжному покажчику.

• Видаляємо перший елемент, застосовуючи покажчик на голову.

• Покажчику на голову списку передаємо адресу другого елемента.

• Повторюємо ці дії, поки список не стане порожнім.

Функція видалення списку може мати такий вигляд:

```
void dropList1(element ** head)
{
    element * p;
    // поки список непорожній
    while(head)
    {
     // збереження посилання на перший елемент, який
видаляємо
        p = *head;
     // встановлення покажчика голови списку на на-
ступний
     //елемент списку
        *head = p->next;
        //видалення першого елемента
        delete(p);
    }
}
або
void dropList2(element ** head)
{
    element * p;
    while(head)
    {
        p = head->next;
        delete(*head);
        *head = p;
    }
}
```

Мовою Kotlin, завдяки автоматичному прибиранню сміття, достатньо «обнулити» посилання на голову та хвіст списку:

```
head = null
tail = null
```

Лабораторний практикум № 12

Мета: набути навичок створення та реалізації програм, що використовують структури лінійного динамічного списку та реалізують операції з ними.

Завдання

Завдання 12.1

Запишіть рядки, які будуть виведені на екран дисплея внаслідок виконання фрагментів, поданих у таблиці 12.1, при наступному початку програми.

Примітка. Замість n підставити номер варіанта за списком групи.

```
#include <iostream>

using namespace std;

struct lnk {
    string name;
    int ph;
    lnk *next;
};

int main() {
    int nr, n;
    int *k, *p;
    lnk *cR, *fst;
    string nAr[3];
    int pAr[3];
```

```
    cin >> n;
    k = &n;
    p = k;
    *p = *p + 2;
    cout << k << " " << *p << endl;
    nAr[0] = "AAA";
    nAr[1] = "BBBB";
    nAr[2] = "CCCC";
    pAr[0] = 2222;
    pAr[1] = 333;
    pAr[2] = 4444;
    nr = sizeof(lnk);
    fst = nullptr;
    for (int i = 0; i < 3; i++) {
        cR = new lnk;
        cR->name = nAr[i];
        cR->ph = pAr[i];
        cR->next = fst;
        fst = cR;
    }
    // код згідно з варіантом додати сюди
}
```

Таблиця 12.1.

Варіанти	Фрагмент

1–5	`fst->next = fst->next->next;`
	`cout << cR->ph << endl;`
	`cR=fst;`
	`while (cR != nullptr){`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr = " << nr;`
	`cout << " " << fst->name << endl;`

Продовж. табл. 12.1

Варіанти	Фрагмент

6–10	`fst=fst->next;`
	`cout << cR->ph << endl;`
	`cR=fst;`
	`while (cR != nullptr) {`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr=" << nr;`
	`cout << " " << fst->name << endl;`

11–15	`lnk * p1=new lnk;`
	`p1->name = nAr[0];`
	`p1->ph=pAr[0];`
	`p1->next=fst->next;`
	`fst->next=p1;`
	`cout << cR->ph << endl;`
	`cR=fst;`
	`for (int i=2;i>0;i--){`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr=" << nr;`
	`cout << " " << fst->name << endl;`

16–20	`lnk * p1=new lnk;`
	`p1->name = nAr[1];`
	`p1->ph=pAr[1];`
	`p1->next=fst->next; fst->next=p1;`
	`cout << cR->ph << endl;`
	`cR=fst;`
	`for (int i=0;i<3;i++){`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr=" << nr;`
	`cout << " " << fst->name << endl;`

Продовж. табл. 12.1

Варіанти	Фрагмент

21–25	`cout << cR->ph << endl;`
	`cR=fst;`
	`for (int i=1;i<3;i++){`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr=" << nr;`
	`cout << " " << fst->name << endl;`

26–30	`cout << cR->ph << endl;`
	`cR=fst;`
	`for (int i=2;i>=0;i--){`
	`cout << cR->name << " " << cR->ph << endl;`
	`cR=cR->next;`
	`}`
	`cout << "nr=" << nr;`
	`cout << " " << fst->name << endl;`

Завдання 12.2 (С++)

Створити лінійний динамічний список на основі структур та файлів, які були створені в процесі виконання лабораторної роботи № 11.

Реалізувати меню для зчитування інформації з файлу у список, виведення елементів списку та виконання операцій згідно з пунктами a-d.

Завдання 12.3 (С++)

Створити функцію для виведення інформації про елемент лінійного списку за покажчиком на нього та виконати пункт (e) з використанням такої функції. Також створити функції додавання запису у список після вказаного елемента та перед указаним елементом. Реалізувати можливість видалення вказаного елемента.

1. **Student:** id, Прізвище, Ім'я, По батькові, Дата народження, Адреса, Телефон, Факультет, Курс, Група.

Запити:

a. Список студентів указаного факультету.

b. Список студентів, які народилися після вказаного року.

c. Список студентів, чиї номери телефонів починаються із вказаної послідовності цифр.

d. Список навчальної групи в алфавітному порядку.

e. Повну інформацію про першого знайденого студента із вказаним прізвищем.

2. **Customer:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Номер банківського рахунку. Запити:

a. Список покупців із вказаним іменем.

b. Список покупців, у яких номер кредитної картки знаходиться в заданому інтервалі.

c. Список покупців, у яких адреса містить у собі вказану послідовність літер (наприклад, назву міста).

d. Список покупців, у яких номер банківського рахунку закінчується на указану цифру.

e. Повну інформацію про покупця із вказаним номером кредитної картки.

3. **Patient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Номер медичної картки, Діагноз. Запити:

a. Список пацієнтів, які мають указаний діагноз.

b. Список пацієнтів, чий номер медичної картки містить указану послідовність цифр.

c. Список пацієнтів, у яких адреса починається із вказаної послідовності символів.

d. Список пацієнтів, номер медичної карти яких знаходиться в заданому інтервалі.

e. Повну інформацію про пацієнта із вказаним прізвищем та номером телефона.

4. **Abiturient:** id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Оцінки. Запити:

a. Список абітурієнтів, які мають незадовільні оцінки.

b. Список абітурієнтів, у яких сума балів вище заданої.

c. Список абітурієнтів, у яких номер телефону починається із заданої послідовності цифр (інші символи номера ігноруються).

d. Вибрати вказану кількість n абітурієнтів, які мають найбільшу суму балів.

e. Повну інформацію про абітурієнта за вказаними прізвищем, ім'ям та по батькові.

5. **Book:** id, Назва, Автор(и), Видавництво, Рік видання, Кількість сторінок, Ціна, Тип палітурки. Запити:

a. Список книг заданого автора.

b. Список книг, що видані вказаним видавництвом.

c. Список книг, кількість сторінок у яких належить указаному діапазону.

d. Список книг, що видані після заданого року.

e. Повну інформацію про книгу із вказаним id.

6. **House:** id, Номер квартири, Площа, Поверх, Кількість кімнат, Вулиця, Тип будівлі, Термін експлуатації. Запити:

a. Список квартир, що мають задану кількість кімнат.

b. Список квартир, що мають указану кількість кімнат і розташованих між указаними поверхами.

c. Список квартир, які експлуатуються не більше *R* (ввести з клавіатури) років, що знаходяться на вказаній вулиці.

d. Список квартир, що мають площу, яка більше заданої.

e. Повну інформацію про квартиру із вказаним id.

7. **Phone:** id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Час міських розмов, Час міжнародних розмов. Запити:

a. Відомості про абонентів, у яких час міських розмов перевищує вказаний.

b. Відомості про абонентів, які користувалися міжнародним зв'язком.

c. Відомості про абонентів, номер кредитної картки яких закінчується на вказану послідовність цифр.

d. Відомості про абонентів в алфавітному порядку.

e. Повну інформацію про абонента із вказаним номером кредитної картки.

8. **Car:** id, Марка, Модель, Рік випуску, Колір, Ціна, Реєстраційний номер. Запити:

a. Список автомобілів заданої марки.

b. Список автомобілів заданої моделі, що експлуатуються більше n років.

c. Список автомобілів указаного кольору, реєстраційний номер яких містить указану послідовність цифр.

d. Список автомобілів указаного року випуску, ціна яких більше вказаної.

e. Повну інформацію про автомобіль із вказаним реєстраційним номером.

9. **Product:** id, Найменування, Тип, Виробник, Ціна, Термін зберігання, Кількість. Запити:

a. Список товарів заданого найменування.

b. Список товарів заданого найменування, ціна яких не більше заданої.

c. Список товарів указаного типу заданого виробника.

d. Список товарів, термін зберігання яких більше заданого.

e. Повну інформацію про товар із вказаним id.

10. **Train:** id, Пункт призначення, Номер поїзда, Час відправлення, Число місць (загальних, плацкарт, купе, люкс). Запити:

a. Список поїздів, які прямують до заданого пункту призначення.

b. Список поїздів, які прямують до заданого пункту призначення та відправляються після вказаної години.

c. Список поїздів, у яких кількість плацкартних місць більше, ніж усіх інших разом.

d. Список поїздів, які відправляються до заданого пункту призначення та мають загальні місця.

e. Повну інформацію про поїзд за його номером.

Завдання 12.4

Розробити програму згідно із завданнями 12.2 та 12.3 мовою Kotlin.

Контрольні питання

1. Що таке структура «лінійний список»?
2. Які лінійні списки існують?
3. Які операції реалізуються з лінійними списками?
4. Як реалізувати додавання елемента в початок, кінець та в середину списка?
5. Як реалізувати вилучення елемента з початку, кінця та середини списка?
6. Як виконати перегляд лінійного списка за допомогою рекурсії?
7. Як виконати перегляд лінійного списка за допомогою ітерації?
8. Яким чином можна реалізувати пошук елемента в лінійному списку?

Тема 13. ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ. СТРУКТУРА «БІНАРНЕ ДЕРЕВО»

Двійкові дерева

Графи й дерева

Спочатку наведемо декілька визначень:

Граф – це непорожня множина точок (вершин) і множина відрізків (ребер), кінці яких належать заданій множині точок [7].

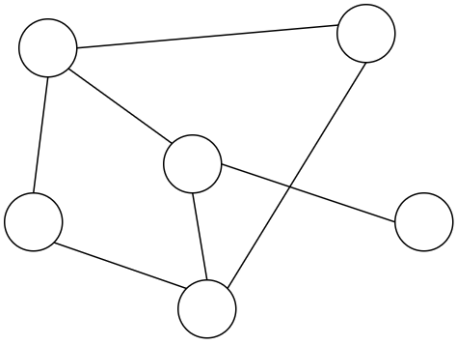


Рис. 13.1. Зображення графа

Якщо на кожному ребрі графа задати напрямок, то він буде орієнтований.

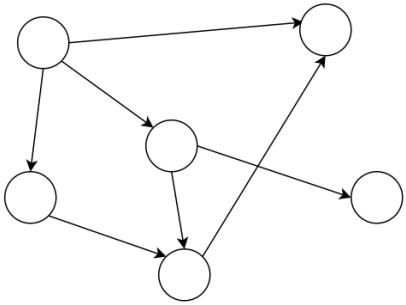


Рис. 13.2. Орієнтований граф

Якщо, рухаючись по ребрах графа в заданому напрямку, можна потрапити із заданої вершини 1 у задану вершину 2, то говорять, що ці **вершини з'єднані шляхом**.

Замкнутий шлях, що складається з різних ребер, називають **циклом**.

Граф називають **зв'язним**, якщо будь-які дві його вершини з'єднані шляхом. Зв'язний граф без циклів називають **деревом**. З кожною вершиною дерева зв'язується скінчена кількість окремих дерев, які називають **піддеревами**.

Схематично дерево можна зобразити таким чином:

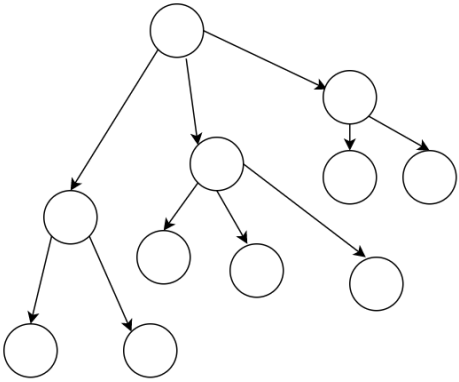


Рис. 13.3. n-арне дерево

Для подальшої роботи з деревами необхідно засвоїти низку понять.

Вершину y, що знаходиться безпосередньо нижче, ніж вершина x, називають безпосереднім нащадком x, а вершину x – предком y.

Якщо вершина не має нащадків, то її називають термінальною вершиною або листом, якщо має – внутрішньою вершиною.

Кількість безпосередніх нащадків внутрішньої вершини називають її ступенем.

Ступенем дерева називають максимальний ступінь усіх вершин.

Двійкове дерево – це такий спосіб подання інформації, за якого однаково ефективно реалізуються всі три основні операції в динамічних структурах: пошуку, запису й видалення інформації. Ця ефективність близька до ефективності дихотомічного пошуку [7].

Двійкове дерево схематично можна зобразити в такий спосіб: є набір вершин, з'єднаних стрілками, з кожної вершини виходить не більше ніж дві стрілки (гілки), спрямовані вліво донизу і/або вправо донизу. Повинна існувати єдина вершина, у яку не входить жодна стрілка – цю вершину називають коренем дерева.

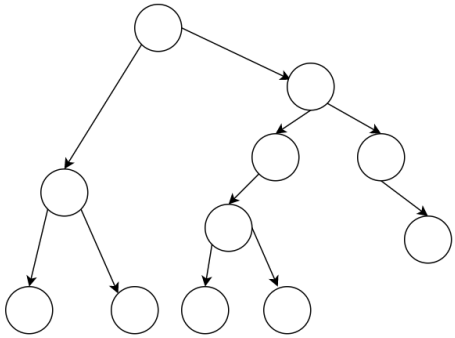


Рис. 13.4. Бінарне (або двійкове) дерево

Будемо вважати, що дані зберігаються в інформаційному полі структури:

```
// C++
struct Node
{
    typeelem data;  // інформаційне поле
    Node * left;    // посилання на ліву гілку
    Node * right;   // посилання на праву гілку
};
// Kotlin
data class Node<T : Comparable<T>?> (
    var data: T,
    var left: Node<T>? = null,
    var right: Node<T>? = null
)
```

Примітка: `<T : Comparable<T>>` означає, що у вершині дерева зберігається значення деякого типу T, що підтримує операцію порівняння (за потреби слід її описати). Приклад опису класу, що підтримує операцію порівняння `compareTo`:

```
data class Student(
    val id:Int,
    var name:String,
    var rating:Double) : Comparable<Student> {

     override fun compareTo(other: Student): Int {
         return if (name == other.name) {
             id - other.id
         } else name.compareTo(other.name)
     }
}
```

Ініціалізація двійкового дерева. Додавання елементів

Функція ініціалізації бінарного дерева нічим не відрізняється від відповідних процедур для списків: покажчика на корінь дерева передається значення `nullptr` (`null`).

```
// C++
node * root = nullptr;

// Kotlin
class Tree<T : Comparable<T>?> {
    var root: Node<T>? = null
    // ...
}
```

Додавання елемента в бінарне дерево можна подати у вигляді функції `addElement`.

Операцію додавання нової вершини у двійкове дерево можна розділити на кілька кроків:

• формування нової вершини;

• пошук вершини, після якої необхідно вставити нову вершину;

• безпосередньо додавання нової вершини в дерево, тобто корегування посилань елементів.

Функція пошуку вершини, після якої необхідно вставити нову вершину, полягає у знаходженні вершини, до якої можна приєднати («підвісити») нову вершину. У випадку надходження запису з новим ключем треба порівняти значення цього ключа із ключами вже наявних вершин. Якщо значення ключа нового елемента менше, ніж значення ключа даного елемента, переходимо на ліву гілку, якщо значення ключа нового елемента більше, ніж значення ключа даного елемента, переходимо на праву гілку. Переміщаючись у такий спосіб по дереву, знаходимо «порожню» вершину, тобто вершину без піддерев, і залежно від результату порівняння ключа в цій вершині із ключем, що надійшов, робимо нову сформовану вершину лівою або правою гілкою дерева.

Наприклад, у нас є послідовність елементів із ключами: 70, 60, 85, 87, 35, 68, 72.

Перший із записів, що надійшли, із ключем 70 робимо коренем дерева.

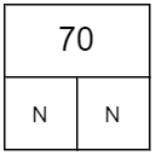


Рис. 13.5. Додавання першого елемента в дерево

Посилання на нижні вершини дорівнюють null.

Наступний ключ 60 менший за 70, виходить, що наступна вершина – ліва для кореня.

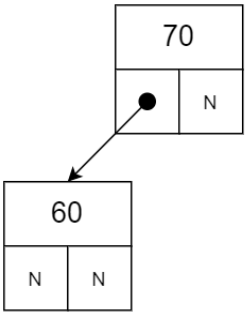


Рис. 13.6. Додавання другого елемента в дерево

Далі 85 – права вершина для кореня.

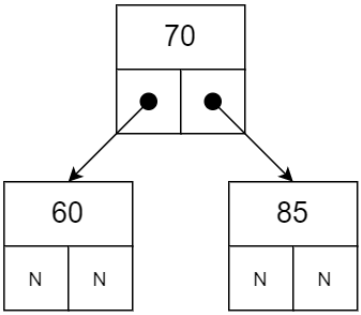


Рис. 13.7. Додавання третього елемента

87 більше ніж 70 та 85. 87 – права вершина для вершини 85.

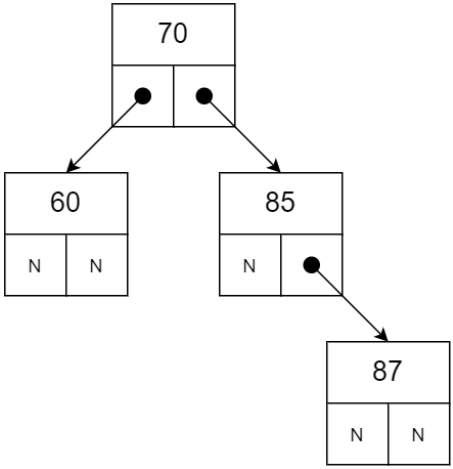


Рис. 13.8. Четверта вершина

Порівнюючи в такий спосіб нові вершини дерева з уже існуючими, одержуємо дерево:

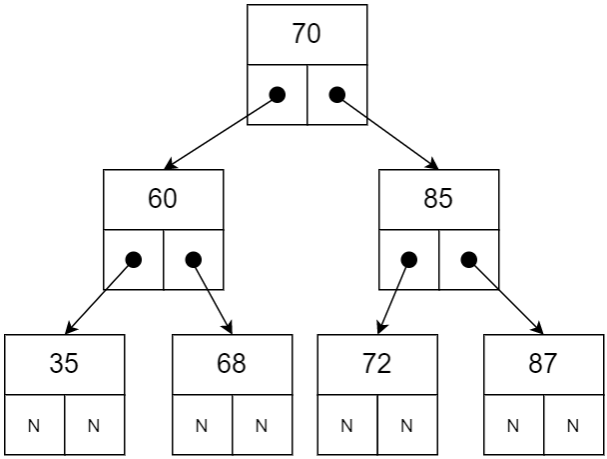


Рис. 13.9. Сформоване в результаті дерево

```
// C++
bool addElement(int value, node **pNode) {
```

```
    if (*pNode == nullptr) {
        node *t = new node;
        t->key = value;
        t->left = nullptr;
        t->right = nullptr;
        *pNode = t;
        return true;
    } else {
        int key = (*pNode)->key;
        if (key == value) return false;
        if (key > value) {
            return addElement(value, &((*pNode)->
left));
        } else {
            return addElement(value, &((*pNode)->
right));
        }
    }
}

// Kotlin
class Tree<T : Comparable<T>?> {

   var root: Node<T>? = null

   fun addElement(value: T): Boolean {
        if (root == null) {
            root = Node(value)
            return true
        }
        return addInSubTree(value, root)
   }

   private fun addInSubTree(value: T, root:
Node<T>?): Boolean {
        if (value == root!!.key) {
            return false
        }
        return if (value!! < root.key) {
```

```
            if (root.left == null) {
                root.left = Node(value)
                true
            } else {
                addInSubTree(value, root.left)
            }
        } else {
            if (root.right == null) {
                root.right = Node(value)
                true
            } else {
                addInSubTree(value, root.right)
            }
        }
    }
    // ...
}
```

Пошук елемента у двійковому дереві

Функція пошуку елемента у двійковому дереві полягає у знаходженні того елемента дерева, значення ключа якого збігається із заданим значенням. Якщо такий елемент знайдено, то функція повертає посилання на нього, інакше – `null` (`nullptr`).

Безпосередньо сам алгоритм пошуку елемента можна описати так:

• значення ключа поточної ланки дерева порівнюють із заданим значенням; якщо значення рівні, алгоритм завершується і функція повертає посилання на поточний елемент;

• залежно від порівняння заданого ключа і ключа поточного елемента, переходимо на ліву / праву гілки (якщо вони існують) і продовжуємо пошук.

```
// C++
node *findElement(int key, node *pNode) {
    if (pNode == nullptr) {
        return nullptr;
```

```
    }
    if (pNode->key == key) {
        return pNode;
    }
    return findElement(key, (pNode->key > key)
               ? pNode->left
               : pNode->right);
}

// Kotlin
class Tree<T : Comparable<T>?> {
    var root: Node<T>? = null
    // ....

    fun find(key: T): Node<T>? {
        return if (root == null) {
            null
        } else findInSubTree(key, root)
    }

    private fun findInSubTree(key: T, root:
Node<T>?): Node<T>? {
        if (root == null || key == root.key) {
            return root
        }
        return if (key!! < root.key) {
            findInSubTree(key, root.left)
        } else {
            findInSubTree(key, root.right)
        }
    }
    // ....
}
```

Обхід та виведення двійкового дерева

Функція обходу (виведення) двійкового дерева на екран, на перший погляд, трохи складна, оскільки елементи розташовуються в нелінійній структурі, для цього необхідно виконати повний обхід дерева.

Для виведення такого роду структур найкраще застосовувати рекурсивний виклик функції виведення одного елемента двійкового дерева, виконуючи для кожної вершини три дії:

- виведення даних, що зберігаються у вузлі;
- обхід лівого піддерева;
- обхід правого піддерева.

Порядок виконання названих дій визначає спосіб обходу дерева. Способи виведення:

- зверху донизу;
- зліва направо;
- знизу нагору.

Функція виведення дерева зліва направо має такий вигляд:

- необхідно спочатку вивести всю ліву гілку дерева;
- потім значення самого кореня дерева;
- а потім усю праву гілку дерева.

Дану функцію необхідно повторити для кожного елемента будь-якої гілки. Таким чином, загальний алгоритм виведення двійкового дерева можна описати в такий спосіб:

- виведення лівої гілки елемента, якщо вона є;
- виведення елемента;
- виведення правої гілки, якщо вона є.

Нижче наведено тексти рекурсивних функцій, що реалізують дану операцію.

```
// C++
void traverseTree(node *pNode) {
    if (pNode != nullptr) {
        traverseTree(pNode->left);
        cout << pNode->key << " ";
        traverseTree(pNode->right);
    }
}

// Kotlin
```

```
class Tree<T : Comparable<T>?> {
    var root: Node<T>? = null
    // ....

    fun traverse() {
        traverse(root)
    }

    private fun traverse(root: Node<T>?) {
        if (root != null) {
            traverse(root.left)
            visit(root)
            traverse(root.right)
        }
    }

    private fun visit(node: Node<T>) {
        println(node.key)
    }
    // ....
}
```

Видалення вузла з двійкового дерева

Безпосереднє видалення запису (вузла) реалізується дуже просто, якщо ця вершина є кінцевою, або з неї виходить тільки одне піддерево, достатньо тільки скорегувати відповідне посилання вершини попередника.

Основні труднощі пов'язані з видаленням вершини, з якої виходять два піддерева. У цьому випадку потрібно знайти відповідну вершину дерева, яку можна було б вставити на місце тієї, що видаляється, причому ця відповідна вершина повинна просто переміщатися.

Така вершина – це або крайній правий елемент лівого піддерева (для досягнення цієї вершини необхідно перейти в наступну вершину по лівій гілці (лівому піддереву), а потім переходити в інші вершини тільки по правій гілці (правому піддереву) доти, поки чергове таке посилання

не буде дорівнювати null), або крайній лівий елемент правого піддерева (для досягнення цієї вершини необхідно перейти в наступну вершину по правій гілці (правому піддереву), а потім переходити в ліві вершини доти, поки чергове таке посилання не буде дорівнювати null).

Очевидно, що знайдені таким чином вершини можуть мати не більше одного піддерева.

Отже, функція видалення із двійкового дерева вершини із заданим ключем повинна розрізняти три випадки:

1. Вершини із заданим ключем у дереві немає.
2. Вершина із заданим ключем має не більше ніж одне піддерево.
3. Вершина із заданим ключем має два піддерева.

Наприклад, розглянемо дерево:

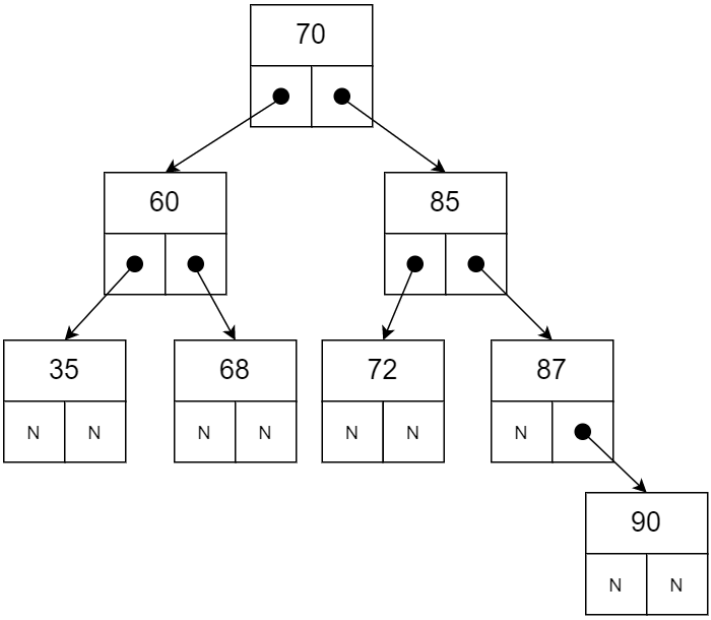


Рис. 13.10. Дерево перед видаленням елементів

Якщо потрібно видалити вершину зі значенням 75, то очевидно, маємо випадок (1) – нічого робити не треба.

Якщо потрібно видалити один з елементів 35, 68, 72 або 90 – ці вершини не мають піддерев, тобто маємо випадок (2) і їх можна просто видалити.

Якщо потрібно видалити елемент 87 – також маємо випадок (2), оскільки ця вершина має одне піддерево, та її можна видалити, замінивши на першу вершину із єдиного піддерева.

Вершини з елементами 60, 85 та 70 мають по два піддерева, отже, маємо випадок (3).

```
// C++
// допоміжна функція пошуку крайнього правого еле-
мента
// у лівому піддереві
void removeEl(node **pNode, node **q) {
    if ((*pNode)->right != nullptr)
        removeEl(&(*pNode)->right, q);
    else {
        (*q)->key = (*pNode)->key;
        *q = *pNode;
        *pNode = (*pNode)->left;
    }
}

// основна функція видалення елемента
void removeElement(int value, node **pNode) {
    if (*pNode != nullptr) {
        if (value < (*pNode)->key)
            removeElement(value, &((*pNode)->left));
        else {
            if (value > (*pNode)->key)
                removeElement(value, &((*pNode)->
right));
            else {
                node *q = *pNode;
                if (q->right == nullptr) *pNode =
q->left;
```

```
                else if (q->left == nullptr) *pNode
= q->right;
                else {
                    removeEl(&(q->left), &q);
                }
                delete q;
            }
        }
    }
}

class Tree<T : Comparable<T>?> {
    var root: Node<T>? = null
    // ....

    fun removeElement(key: T) {
        removeElement(key, root)
    }

    private fun removeElement(key: T, node:
Node<T>?) : Node<T>? {
        var root = node
        if (root == null) return root
        if (key!! > root.key) {
            root.right = removeElement(key, root.
right)
        } else if (key < root.key) {
            root.left = removeElement(key, root.
left)
        } else {
            if (root.left == null && root.right ==
null) {
                root = null
            } else if (root.right != null) {
                root.key = successor(root)
                root.right = removeElement(root.key,
root.right)
            } else {
                root.key = predecessor(root)
```

```
                root.left = removeElement(root.key,
root.left)
            }
        }
        return root
    }

    /*
       Пошук безпосередньо наступного значення
       Крайній лівий елемент у правому піддереві
    */
    private fun successor(node: Node<T>) : T {
        var root = node
        root = root.right!!
        while (root.left != null) {
            root = root.left!!
        }
        return root.key
    }

    /*
       Пошук безпосередньо попереднього значення
       Крайній правий елемент у лівому піддереві
    */
    private fun predecessor(node: Node<T>) : T {
        var root = node
        root = root.left!!
        while (root.right != null) {
            root = root.right!!
        }
        return root.key
    }
    // ....
}
```

Примітка. Зверніть увагу, що код мовою Kotlin дещо складніший, через відсутність механізмів роботи з покажчиками, і хоча здебільшого покажчики легко замінюються на безпечні посилання, деякі складнощі все одно залишають-

ся. Наприклад, неможливо отримати покажчик на покажчик і через те, доводиться писати більш складний код. Але це компенсується тим, що в Kotlin (як і в Java) є автоматичне «прибирання сміття» – вивільнення пам'яті, яку займали видалені об'єкти.

Лабораторний практикум № 13

Мета: набути навичок створення та реалізації програм, що використовують структуру «Бінарне дерево» та реалізують операції з ними.

Завдання

Примітка: Кожне завдання реалізувати з використанням мов програмування C/C++ та Kotlin.

Завдання 13.1

Створити програму, що реалізує структуру «Бінарне дерево» для зберігання та оброблення структур та файлів, які були створені в процесі виконання лабораторної роботи № 12.

Завдання 13.2

Створити функцію для виведення інформації про елемент за вказівником на нього та виконати пункт (d) з використанням такої функції. Також створити функції додавання запису в дерево.

1. **Student**: id, Прізвище, Ім'я, По батькові, Дата народження, Адреса, Телефон, Факультет, Курс, Група. Створити бінарне дерево пошуку структур. Вивести:

a. Список студентів указаного факультету в порядку зростання дати народження.

b. Список студентів, які народилися після вказаного року.

c. Список навчальної групи в алфавітному порядку.

d. Повну інформацію про першого знайденого студента із вказаним прізвищем.

2. **Customer**: id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Номер банківського рахунку. Створити бінарне дерево пошуку структур.

Вивести:

a. Список покупців в алфавітному порядку.

b. Список покупців, у яких номер кредитної картки знаходиться в заданому інтервалі.

c. Список покупців, у яких номер банківського рахунку закінчується на вказану цифру.

d. Повну інформацію про покупця із вказаним номером кредитної картки.

3. **Patient**: id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Номер медичної картки, Діагноз. Створити бінарне дерево пошуку структур.

Вивести:

a. Список пацієнтів, які мають указаний діагноз у порядку зростання номерів медичної картки.

b. Список пацієнтів, чий номер медичної картки при діленні на 7 дасть указану остачу.

c. Список пацієнтів, номер медичної карти яких знаходиться в заданому інтервалі.

d. Повну інформацію про пацієнта із вказаним прізвищем та номером телефона.

4. **Abiturient**: id, Прізвище, Ім'я, По батькові, Адреса, Телефон, Оцінки. Створити бінарне дерево пошуку структур.

Вивести:

a. Список абітурієнтів, які мають незадовільні оцінки в алфавітному порядку.

b. Список абітурієнтів, у яких сума балів вище заданої.

c. Вибрати вказану кількість n абітурієнтів, які мають найбільшу суму балів.

d. Повну інформацію про абітурієнта за вказаними прізвищем, ім'ям та по батькові.

5. **Book**: id, Назва, Автор(и), Видавництво, Рік видання, Кількість сторінок, Ціна, Тип палітурки. Створити бінарне дерево пошуку структур.

Вивести:

a. Список книг заданого автора в порядку спадання року видання.

b. Список книг, що видані вказаним видавництвом.

c. Список книг, що видані після заданого року.

d. Повну інформацію про книгу із вказаним id.

6. **House**: id, Номер квартири, Площа, Поверх, Кількість кімнат, Вулиця, Тип будівлі, Термін експлуатації. Створити бінарне дерево пошуку структур. Вивести:

a. Список квартир, що мають задану кількість кімнат у порядку зростання терміну експлуатації.

b. Список квартир, що мають указану кількість кімнат і розташованих між указаними поверхами.

c. Список квартир, які мають площу, що більше заданої.

d. Повну інформацію про квартиру із вказаним id.

7. **Phone**: id, Прізвище, Ім'я, По батькові, Адреса, Номер кредитної картки, Дебет, Кредит, Час міських та міжнародних розмов. Створити бінарне дерево пошуку структур.

Вивести:

a. Відомості про абонентів, у яких час міських розмов перевищує вказаний у порядку алфавіту.

b. Відомості про абонентів, які користувались міжміським зв'язком.

c. Відомості про абонентів в алфавітному порядку.

d. Повну інформацію про абонента із вказаним номером кредитної картки.

8. **Car**: id, Марка, Модель, Рік випуску, Колір, Ціна, Реєстраційний номер. Створити бінарне дерево пошуку структур.

Вивести:

a. Список автомобілів заданої марки, упорядкований за зростанням реєстраційного номера.

b. Список автомобілів заданої моделі, які експлуатуються більше n років.

c. Список автомобілів указаного року випуску, ціна яких більше вказаної.

d. Повну інформацію про автомобіль із вказаним реєстраційним номером.

9. **Product**: id, Найменування, Тип, Виробник, Ціна, Термін зберігання, Кількість. Створити бінарне дерево пошуку структур.

Вивести:

a. Список товарів заданого найменування, упорядкований за зростанням ціни.

b. Список товарів заданого найменування, ціна яких не більше заданої.

c. Список товарів, термін зберігання яких більше заданого.

d. Повну інформацію про товар із вказаним id.

10. **Train**: id, Пункт призначення, Номер поїзда, Час відправлення, Число місць (загальних, плацкарт, купе, люкс). Створити бінарне дерево пошуку структур.

Вивести:

a. Список поїздів, що прямують до заданого пункту призначення, упорядкований за часом відправлення.

b. Список поїздів, які прямують до заданого пункту призначення та відправляються після вказаної години.

c. Список поїздів, які відправляються до заданого пункту призначення та мають загальні місця.

d. Повну інформацію про поїзд за його номером.

Завдання 13.3 (додаткове)

Реалізувати можливість видалення вказаного елемента.

Контрольні питання

1. Що таке абстрактна структура даних «дерево»?
2. Яка особливість структури «двійкове дерево»?
3. Які способи обходу двійкового дерева існують?
4. Як відбувається пошук елемента у двійковому дереві?
5. Як відбувається додавання елемента у двійкове дерево?
6. Як відбувається вилучення елемента з двійкового дерева?

ВИСНОВКИ

У посібнику розглянуто основні засоби програмування мовами Kotlin та C++. Виконавши всі завдання лабораторних практикумів, ви ознайомилися з типовими задачами, які розв'язує програміст у роботі. Ознайомились із абстрактними структурами даних «список» та «бінарне дерево», основами об'єктно-орієнтованого підходу до складання програм. Ці знання та вміння стануть у нагоді під час вивчення дисциплін «Структури та організація даних», «Об'єктно-орієнтоване програмування», «Організація баз даних та знань» та інших.

СПИСОК ЛІТЕРАТУРИ

1. Жемеров Д., Исакова С. Kotlin в действии / пер. с англ. А. Н. Киселев. Москва : ДМК Пресс, 2018. 402 с.: ил.

2. Введение в язык Котлин [Електронний ресурс] : [Вебсайт]. – Електронні дані. – Coursera 2021. – Режим доступу: https://www.coursera.org/learn/vvedenie-v-yazyk-kotlin (дата звернення 15.05.2021) – Назва з екрана.

3. Irina Galata, Joe Howard, Ellen Shapiro. Kotlin Apprentice. – Razeware LLC, 2019. 474 p.

4. Kotlin docs [Електронний ресурс] : [Вебсайт]. – Електронні дані. – JetBrains 2021. – Режим доступу: https://kotlinlang.org/docs/home.html (дата звернення 15.05.2021) – Назва з екрана.

5. Павловская Т. А. C/C++. Программирование на языке высокого уровня. СПб.: Питер, 2003. 461 с.: ил.

6. Михелєв І. Л., Слободян С. О. Основи інформаційних технологій : навчальний посібник. Миколаїв : НУК, 2014. 164 с.

7. Томас Г. Кормен, Чарлз Е. Лейзерсон, Роналд Л. Рівест, Кліфорд Стайн. Вступ до алгоритмів. Київ : К. І. С., 2019. 1288 с.

8. Скин Джош, Гринхол Дэвид Kotlin. Программирование для профессионалов. СПб.: Питер, 2020. 464 с.: ил. (Серия «Для профессионалов»).

ЗМІСТ

Навчальне видання

БЕРКУНСЬКИЙ Євген Юрійович
ПАВЛЕНКО Альона Юріївна

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ МОВАМИ KOTLIN, C/C++

Навчальний посібник

Комп'ютерна правка й коректура *О. Г. Костенко*
Комп'ютерне верстання *В. В. Москаленко*

Формат 60×84/16. Ум. друк. арк. 14,9. Тираж 100 прим. Вид. № 35. Зам. № 2209-29.
Видавець і виготівник Національний університет кораблебудування
імені адмірала Макарова
просп. Героїв України, 9, м. Миколаїв, 54025
E-mail : publishing@nuos.edu.ua
Свідоцтво суб'єкта видавничої справи ДК № 6402 від 19.09.2018 р.