МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет імені Тараса Шевченка

О. В. Прокопенко, М. О. Попов, Г. Л. Чумак

Мова програмування C/C++. Практикум

Навчальний посібник

Київ – 2024

УДК 004.43 (075.8)
ББК 32.973.26-018.1

Рекомендовано до друку Вченою радою
навчально-наукового інституту високих технологій
Київського національного університету імені Тараса Шевченка
(протокол № 12 від 4 червня 2024 року)

Рецензенти:
Войтешенко І. С. – кандидат фіз.-мат. наук, доцент;
Іванов І. І. – кандидат фіз.-мат. наук, доцент.

Прокопенко О. В.
Мова програмування C/C++. Практикум: навчальний посібник / О. В. Прокопенко, М. О. Попов, Г. Л. Чумак. – К.: Київський національний університет імені Тараса Шевченка, 2024. – 375 с.

Навчальний посібник написаний на основі циклів лабораторних робіт з курсів «Програмування», «Мова програмування C/C++», які проводились для студентів навчально-наукового інституту високих технологій та механіко-математичного факультету Київського національного університету імені Тараса Шевченка.

Книга охоплює ключові теми, необхідні для швидкого опанування на практиці мистецтва програмування мовами C/C++. Матеріал посібника організовано у вигляді 10 лабораторних робіт, кожна з яких включає в себе короткі теоретичні відомості, приклади виконання завдань, перелік індивідуальних завдань для студентів, а також контрольні запитання. Навчальний посібник написано за принципом «від простого – до складного». Він включає в себе більше 60 програм, які демонструють основні підходи до розв'язання практичних задач за допомогою мов C/C++.

Для студентів природничо-математичних, технічних і комп'ютерних спеціальностей закладів вищої освіти, викладачів, інженерів і всіх тих, хто самостійно опановує мови C/C++.

ЗМІСТ

Вступ

Мови програмування[1] C та C++ існують вже давно (першій виповнилось більше 50 років, іншій – близько 40). Проте і сьогодні вони залишаються популярними та затребуваними[2], тому що є гнучкими у використанні і дозволяють створювати програмний код з дуже високою продуктивністю[3] (швидкодією) при достатньо малих затратах часу і ресурсів.

Знання мов C/C++ важливе для розв'язання багатьох практичних задач, що часто зустрічаються в різних областях науки і техніки. Не менш важливе значення цих мов і для опанування методології сучасного програмування – ці мови дозволяють реалізувати всі основні шаблони та технології програмування, відомі людству. До того ж, як показує досвід, знання мов C/C++ полегшує опанування інших мов програмування. Саме тому вивчення мов програмування C/C++ є важливим елементом підготовки фахівця в галузі природничо-математичних, технічних та комп'ютерних наук.

Мета цієї книги – допомогти навчитися ефективно використовувати мови C/C++ для розв'язання практичних задач. Основна частина навчального посібника включає в себе описи 10 лабораторних робіт, кожна з яких містить теоретичні відомості, приклади виконання завдань, перелік індивідуальних завдань для студентів, а також контрольні запитання. Крім того, у посібнику наводиться інформація щодо налаштувань інструментів розробника C/C++.

Навчальний посібник організовано таким чином, щоб по можливості спростити опанування мов C/C++ і в той же час зробити цей процес доволі швидким. Для цього в посібник включено велику кількість оригінальних програм, які демонструють як типові, так і нестандартні підходи до розв'язання практичних задач за допомогою мов C/C++.

Лабораторні роботи рекомендується виконувати послідовно, спочатку роботи, що переважно стосуються мови програмування C, і лише потім роботи, що стосуються C++. Тематика лабораторних робіт підібрана таким чином, щоб максимально спростити вивчення синтаксису мов C/C++ і одночасно з цим проілюструвати найбільш типові для C/C++ концепції та методи програмування.

При написанні посібника автори переважно орієнтувались на класичні (доволі консервативні) стандарти мов C/C++. Для мови програмування C це стандарт C99, а для мови програмування C++ це стандарт C++98. Разом з тим, в книзі коротко розглянуто деякі особливості більш сучасних стандартів мов

---

[1] Для полегшення сприйняття тексту посібника, словосполучення «мова програмування» або «мови програмування» перед C, C++ або C/C++ іноді будемо пропускати.

[2] У 2020–2023 рр. мови програмування C та C++ входили до трійки найбільш популярних мов світу. Мова C обіймала 2 місце, а C++ – 3 місце згідно міжнародного рейтингу TIOBE programming community index.

[3] Більшість існуючих на даний момент операційних систем, драйверів пристроїв, програм керування медичним обладнанням та озброєнням, програм моделювання та роботи з графікою, офісних програм, складних комп'ютерних ігор тощо написані саме мовами C або C++.

програмування C/C++.

Програми, наведені в посібнику, розроблялись в безкоштовному програмному середовищі Microsoft Visual Studio Community 2022. Іноді робота цих програм додатково перевірялась у безкоштовних середовищах Embarcadero Dev-C++ та/або Code::Blocks. Відповідні тексти програм за наявності мінімальних змін (або, навіть, без змін) мають успішно оброблятись в будь-якому з цих середовищ[1].

Основні позначення

IDE – Integrated Development Environment (Інтегроване середовище розробки програм)
ОС – операційна система
ООП – об’єктно-орієнтоване програмування

[1] Кожен з інструментів розробника C/C++ має певні особливості. Наприклад, Microsoft Visual Studio Community 2022 дозволяє включати в текст програми деякі програмні конструкції, специфічні саме для Microsoft Visual C++. При цьому обробка такої програми в Dev-C++ може призвести до появи повідомлень про помилки, оскільки пакет Dev-C++ нічого не знає про програмні конструкції, специфічні для Microsoft Visual C++. Та ж сама поведінка може спостерігатись і при обробці тексту програми в Code::Blocks.

Підготовка до виконання лабораторних робіт

Процес розробки програм на C/C++

Розробка програми на C/C++ передбачає виконання наступних обов'язкових кроків (рис. 1):

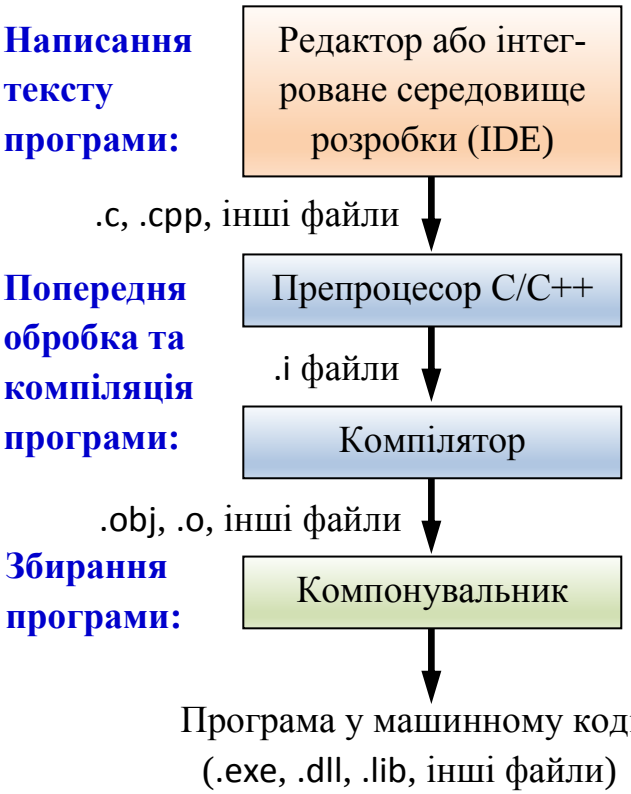


Рис. 1

Написання тексту програми. У найпростішому випадку, програма мовою C/C++ – це звичайний фрагмент тексту, який типово зберігається у файлі з розширенням .c або .cpp. Цей файл можна створити і редагувати у будь-якому текстовому редакторі, наприклад, за допомогою редактора Блокнот, що входить до складу операційної системи (ОС) Windows.

При використанні текстового редактора необхідно враховувати, що текст програми має зберігатись у файлі .c/.cpp безпосередньо як «сирий» текст (формат .txt файлів). Для інших форматів збереження даних, наприклад, у форматах .doc та .docx, характерних для Microsoft Word, текст програми, як правило, буде оброблятись невірно.

Попередня обробка та компіляція програми. Після того як текст програми написаний, його слід обробити, щоб отримати т. зв. об'єктний (машинний) код. Цей процес називається ***компіляцією***. Він виконується ***компілятором (compiler)*** – спеціальною програмою, яка перетворює текст написаної мовою C/C++ програми в об'єктний (машинний) код.

Сучасні компілятори виконують процес попередньої обробки та компіляції (часто говорять просто процес компіляції) програми принаймні в два етапи[1]. Спочатку викликається т. зв. ***препроцесор C/C++*** (***C/C++ preprocessor***) – спеціальна програма, яка обробляє та змінює текст написаної мовою C/C++ програми перед її подальшою компіляцією. В результаті роботи препроцесора C/C++ створюється модифікований текст програми (іноді він зберігається як

[1] Залежно від виробника та версії компілятора таких етапів може бути більше двох. Наприклад, при створенні оптимізованого коду, компілятор може реалізовувати додатковий етап оптимізації коду, під час якого певні фрагменти програми можуть проходити повторну компіляцію.

проміжний файл з розширенням .i). Потім цей модифікований текст обробляється власне компілятором, який створює файл з розширенням .o або .obj, що містить об'єктний код програми. Як правило, ім'я об'єктного .o/.obj файлу співпадає з ім'ям вихідного .c/.cpp файлу. Таким чином, наприклад, при компіляції файлу lab01.cpp створюється об'єктний файл lab01.obj або lab01.o і можливо проміжний файл lab01.i.

Збирання або компонування програми. Після завершення компіляції програми потрібно запустити ***компонувальник*** або ***редактор зв'язків (linker)*** – спеціальну програму, яка об'єднує фрагменти об'єктного коду (типово файли з розширенням .o, .obj, .lib, .a) в єдиний машинний код кінцевої програми.

Виклик компонувальника є обов'язковим кроком, оскільки самого по собі об'єктного .o/.obj файлу, створеного компілятором з .c/.cpp файлу, не достатньо для створення кінцевої програми на C/C++. Справа в тому, що у тексті вихідної програми, як правило, використовуються різноманітні типи даних та функції, реалізовані в зовнішніх програмних бібліотеках, зокрема, в стандартних бібліотеках C/C++. Отже, об'єктний код, створений компілятором, не може успішно виконуватись, поки до цього коду не підключено потрібний об'єктний код із використаних зовнішніх бібліотек. Крім того, для роботи більшості програм на C/C++ необхідний спеціальний фрагмент об'єктного коду – т. зв. код запуску або код ініціалізації (startup code). Для створення працездатної кінцевої програми, цей код запуску необхідно під'єднати до об'єктного коду, створеного компілятором з .c/.cpp файлу (що, власне, й робить компонувальник).

ІНСТРУМЕНТИ РОЗРОБНИКА C/C++

Раніше розглядався лише найпростіший випадок, коли текст програми мовою C/C++ міститься в одному єдиному файлі, наприклад, у файлі lab01.cpp. При компіляції цього файлу має створитися об'єктний файл lab01.obj, а після виклику компонувальника – виконуваний файл або файл застосунку lab01.exe.

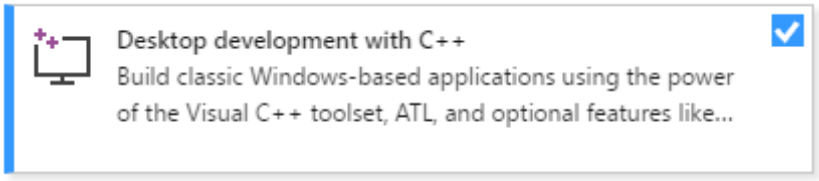
Насправді ж, сучасні програми мовою C/C++ можуть складатись з сотень і тисяч файлів і містити в собі тисячі і навіть мільйони рядків коду. Крім того, в кінцевий бінарний файл програми (застосунку, динамічної або статичної бібліотеки тощо) часто включаються різноманітні дані, що використовуються програмою, але не мають прямого відношення до її коду – піктограми, курсори, графічні зображення, текстові фрагменти, аудіозаписи, відеоролики тощо. Все це помітно ускладнює контроль за складовими програми на C/C++ та їх належну обробку. Тому для полегшення розробки програм, використовують ***інструменти розробника C/C++***. Як правило, такий інструмент являє собою ***інтегроване середовище розробки (Integrated Development Environment,***

IDE), яке має графічний інтерфейс, дозволяє легко контролювати розробку програми і автоматизує її компіляцію та збирання.

Робота сучасних IDE ґрунтується на використанні проєктів. ***Проєкт*** – це файл (або декілька файлів), який містить всю інформацію, необхідну для створення кінцевої програми на C/C++. Проєкт включає в себе інформацію про всі вихідні файли, необхідні для створення програми, інформацію про особливості побудови програми (тип процесора, тип операційної системи, наявність оптимізації коду тощо), а також принаймні повний набір опцій, які мають використовуватись під час роботи препроцесора C/C++, компілятора та компонувальника.

Відомо більше десятка інтегрованих середовищ розробки (IDE). Кожен IDE, як правило, реалізує в собі потенціал текстового редактора з розпізнаванням синтаксису C/C++, автоматизує процеси компіляції та збирання програм C/C++, дозволяє проводити відлагодження програм, вимірювання їх швидкодії тощо. Стисло розглянемо можливості трьох безкоштовних IDE, які можна використати для виконання лабораторних робіт та професійного програмування на C/C++.

Microsoft Visual Studio Community 2022. Найбільш популярний і повнофункціональний IDE на даний момент. Дозволяє розробляти програми довільної складності різними мовами програмування (C/C++, C#, F#, JavaScript, Python, Visual Basic тощо). Має найбільшу кількість вбудованих інструментів для професійного розроблення, відлагодження, оптимізації коду програм. Версія Community даного IDE безкоштовна для використання в навчальному процесі. Можливим недоліком цього IDE є високі системні вимоги та його орієнтованість на ОС Windows. Також на момент видання книги був відсутній україномовний варіант інтерфейсу IDE.



Зауваження: інсталяція всіх компонент Microsoft Visual Studio Community 2022 не є обов'язковою. Для виконання лабораторних робіт з розробки програм на C/C++достатньо вибрати лише компонент Розробка класичних застосунків на C++ (Desktop development with C++). Усі інші компоненти встановлюються за бажанням.

Де знайти: https://visualstudio.microsoft.com/downloads/

Embarcadero Dev-C++. Простий у використанні безкоштовний IDE з базовими можливостями розробки програм на C/C++. Ідеальний варіант для використання на «слабких» комп'ютерах з малою швидкодією і малим об'ємом оперативної пам'яті. Підтримує україномовний інтерфейс і орієнтований на створення програм для ОС Windows.

Де знайти: https://sourceforge.net/projects/embarcadero-devcpp/

Code::Blocks. Безкоштовний IDE з можливостю розробки програм

мовами C/C++ та Fortran для різних операційних систем. За своїм функціоналом займає проміжне місце між Microsoft Visual Studio Community 2022 та Embarcadero Dev-C++. Існують реалізації цього IDE для різних операційних систем (Windows, Linux, Mac OS X).

Де знайти: https://www.codeblocks.org/downloads/binaries/

Виконувати лабораторні роботи з програмування на C/C++ рекомендується в одному з доступних IDE. Для цього необхідно:

- запустити IDE і створити в ньому новий проєкт (див. далі);
- створити файл (або файли), в яких буде міститись текст програми і під'єднати його (їх) до проєкту;
- написати текст програми у під'єднаних до проєкту файлах;
- провести компіляцію і збирання програми.

СТВОРЕННЯ ПРОЄКТУ В IDE ТА ДОСТУП ДО ЙОГО НАЛАШТУВАНЬ

Розглянемо створення та доступ до налаштувань проєкту в трьох згаданих вище IDE, а саме Microsoft Visual Studio Community 2022, Embarcadero Dev-C++ та Code::Blocks.

Microsoft Visual Studio Community 2022

Загальний вигляд вікна IDE показано на рис. 2 (цей вигляд може дещо відрізнятись залежно від налаштувань IDE).

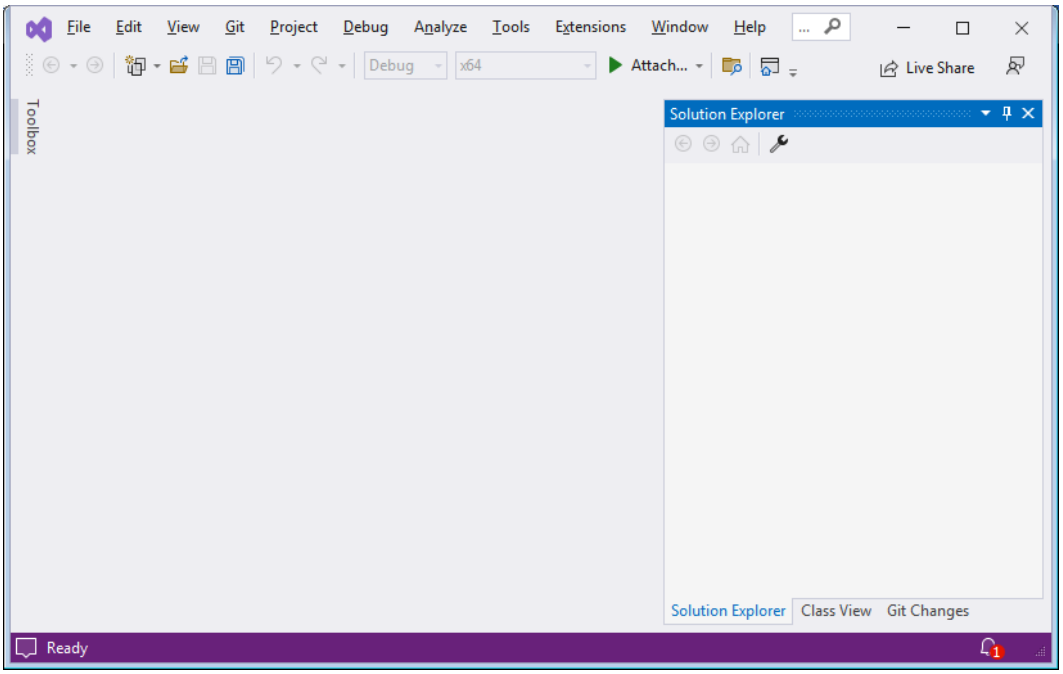


Рис. 2

Для створення нового проєкту в меню File обираємо елемент New. Відкривається випадаюче меню, в якому обираємо елемент Project…. Відкривається діалог Create a new project (рис. 3).

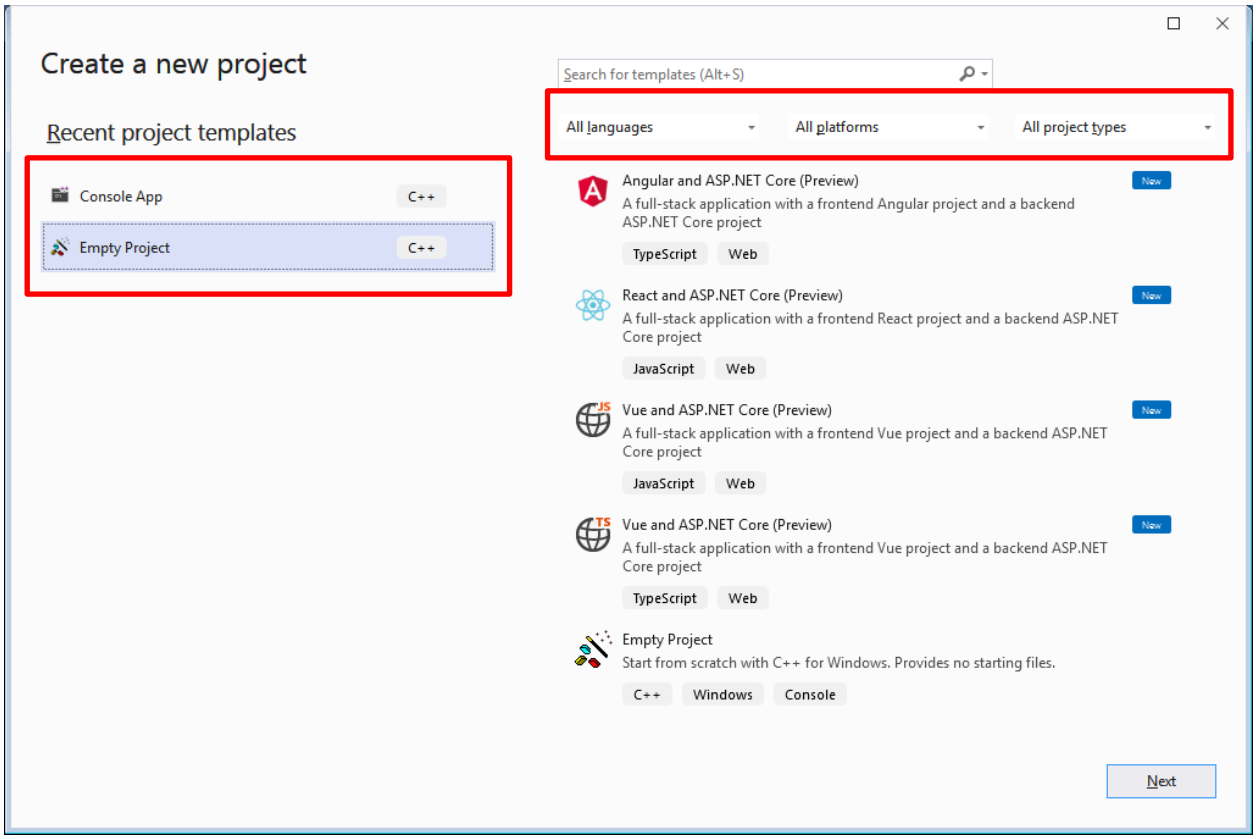


Рис. 3

В лівій частині цього діалогу наводиться інформація щодо нещодавно використаних типів проєктів. На рис. 3 це Console App та Empty Project. Вибір будь-якого з цих варіантів дозволяє відразу ж створити відповідний проєкт.

Розглянемо тепер як створити проєкт, явно задаючи всі його властивості.

У діалозі Create a new project є три випадаючих списки (див. рис. 3): перший задає мову програмування (зараз вибрано All languages), другий задає ОС, для якої буде створюватись програма (зараз вибрано All platforms), а третій список визначає тип проєкту (зараз вибрано All project types). Вибираємо в першому списку мову програмування C++. У другому списку, наприклад, обираємо ОС Windows. В останньому третьому списку обираємо Console[1]. Після цього перелік доступних типів проєктів, наведений нижче випадаючих списків, зменшується. У цьому переліку обираємо тип проєкту Console App для

[1] Цей тип проєкту відповідає т. зв. ***консольним застосункам*** – програмам, що не мають повноцінного графічного інтерфейсу і дозволяють вводити/виводити дані в текстовому режимі через спеціальне вікно – т. зв. системну ***консоль***. Розробка консольних застосунків на C/C++ є значно простішою за розробку програм з графічним інтерфейсом. Тому цей тип проєкту добре підходить для вивчення програмування на C/C++.

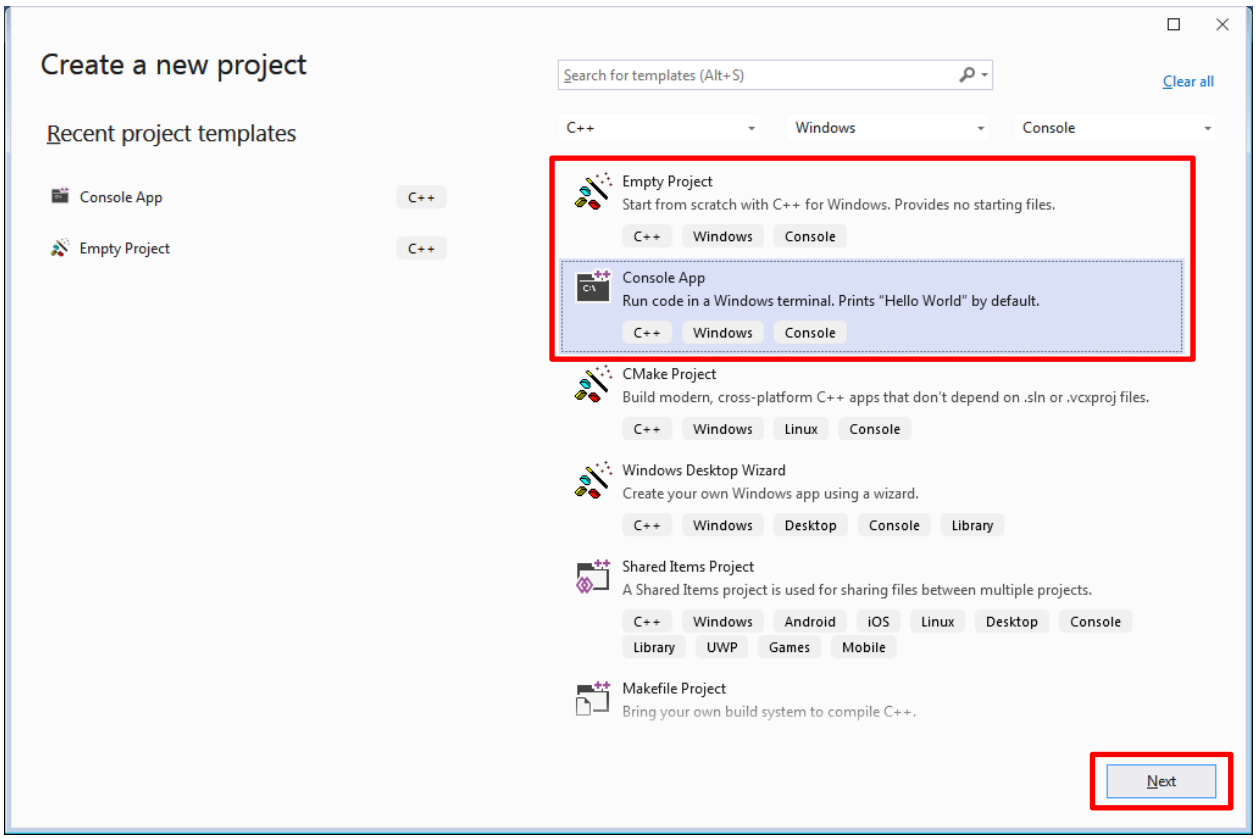


Рис. 4

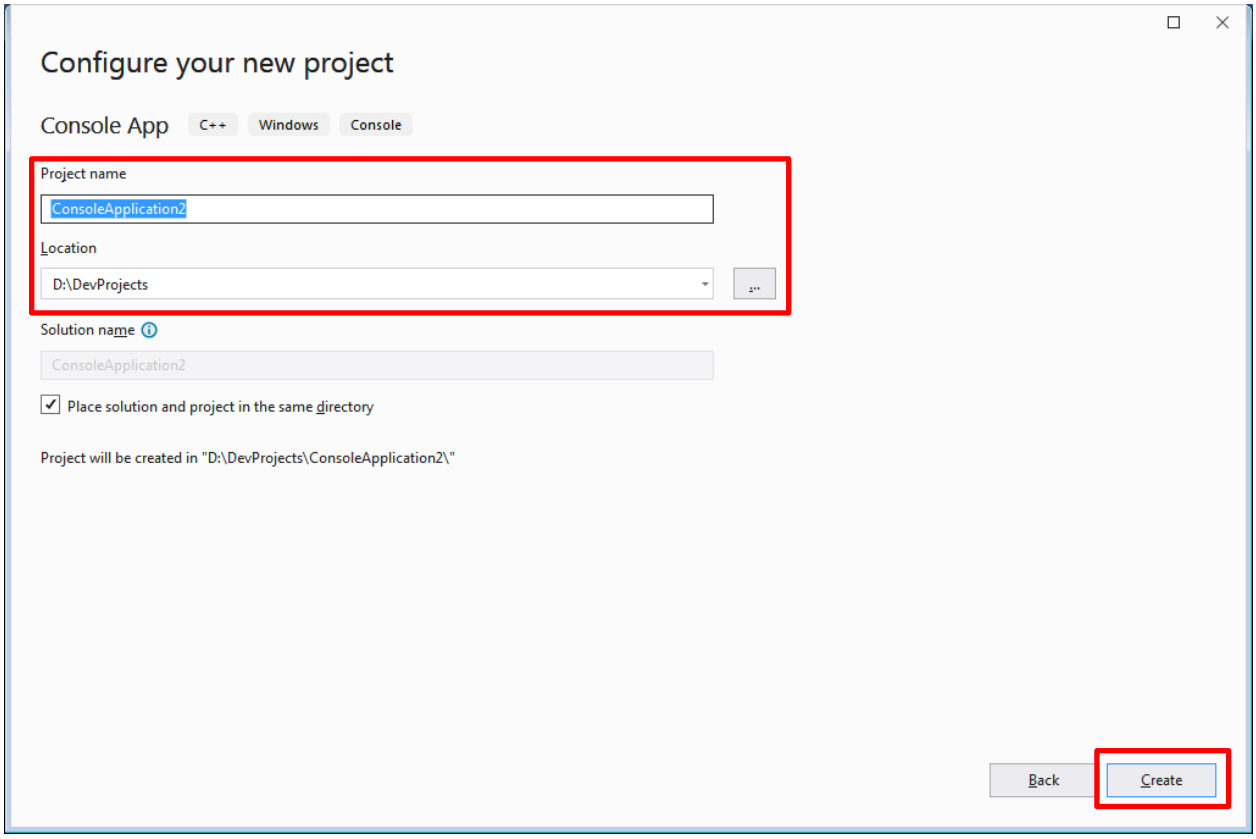


Рис. 5

створення консольного застосунку з мінімальним кодом (рис. 4). Якщо ж

обрати перший в переліку тип проєкту Empty Project, буде створений пустий проєкт, який не містить жодного .c/.cpp файлу. Зробивши вибір одного з вказаних типів проєктів, натискаємо кнопку Next.

Відкривається діалог Configure your new project, в якому потрібно задати властивості проєкту, що створюється (рис. 5). В цьому діалозі необхідно задати ім'я проєкту (поле Project name, зараз містить ConsoleApplication2) та шлях, за яким має бути створена папка з файлами проєкту (поле Location, зараз містить D:\DevProjects). Увівши потрібні дані в поля Project name та Location, натискаємо кнопку Create, що приводить до створення проєкту. Вигляд вікна IDE відразу після створення проєкту показано на рис. 6.

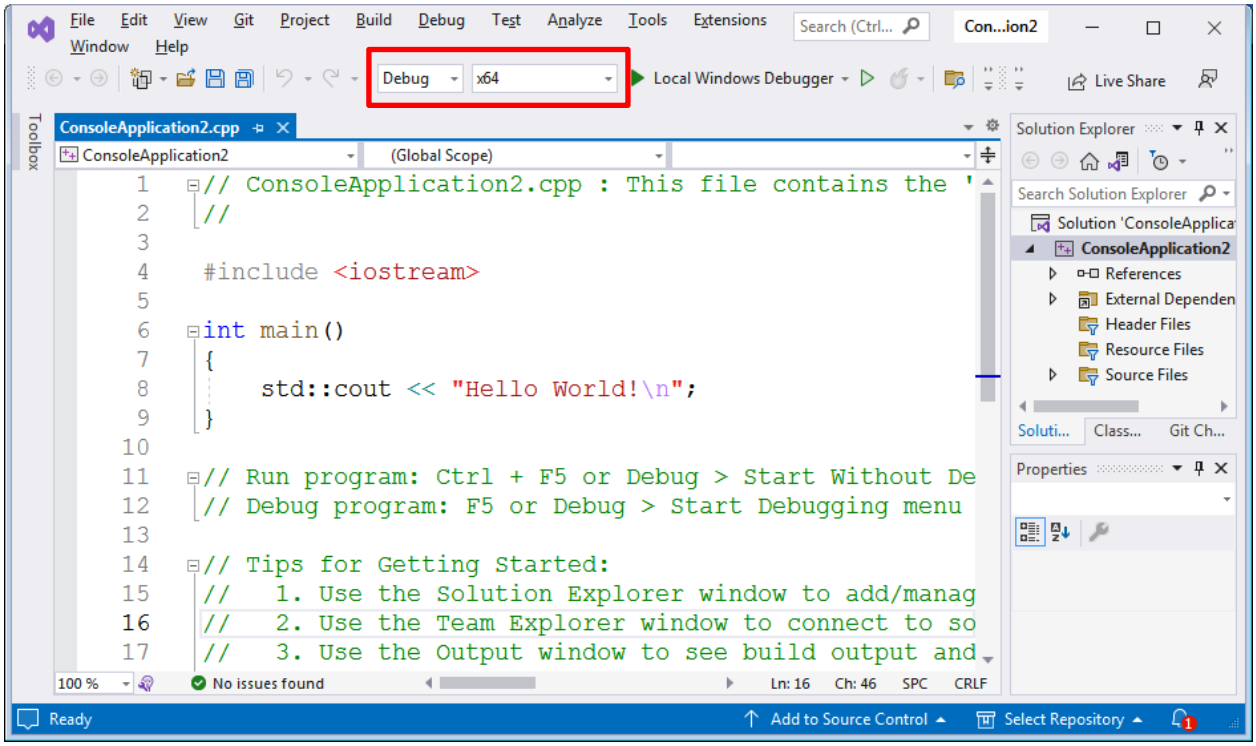


Рис. 6

Створений проект за замовчуванням є налагоджувальною (debug) версією програми для роботи на 64-розрядних процесорах (випадаючі списки з вибраними елементами Debug та x64 згори вікна IDE на рис. 6).

При виконанні лабораторних робіт рекомендується обирати саме налагоджувальну (debug) версію програми, оскільки тоді в коді програми будуть автоматично виконуватись додаткові перевірки даних, буде проводитись виведення діагностичних повідомлень при виникненні помилок, буде можливість застосовувати ефективні засоби пошуку та обробки помилок тощо. Все це разом помітно полегшує розробку та відлагодження кінцевої програми.

Для сучасних комп'ютерів варто обирати апаратну платформу як x64, що відбувається за замовчуванням. Для більш застарілих комп'ютерів можна вибирати x86, при цьому машинний код програми буде створюватись для 32-розрядних або більш кращих процесорів.

Створений проєкт має величезну кількість опцій-параметрів (їх більше сотні). Доступ до цієї інформації можна отримати з вікна Solution Explorer (якщо це вікно закрите, обираємо пункт головного меню IDE View → Solution Explorer, щоб його відкрити). У вікні Solution Explorer вибираємо вузол проєкту (другий елемент-вузол з ім'ям ConsoleApplication2 – див. рис. 6) і натискаємо праву кнопку миші. У меню, що з'явилось, обираємо останній пункт Properties, що приводить до відкриття діалогу з інформацією про властивості проєкту (рис. 7).

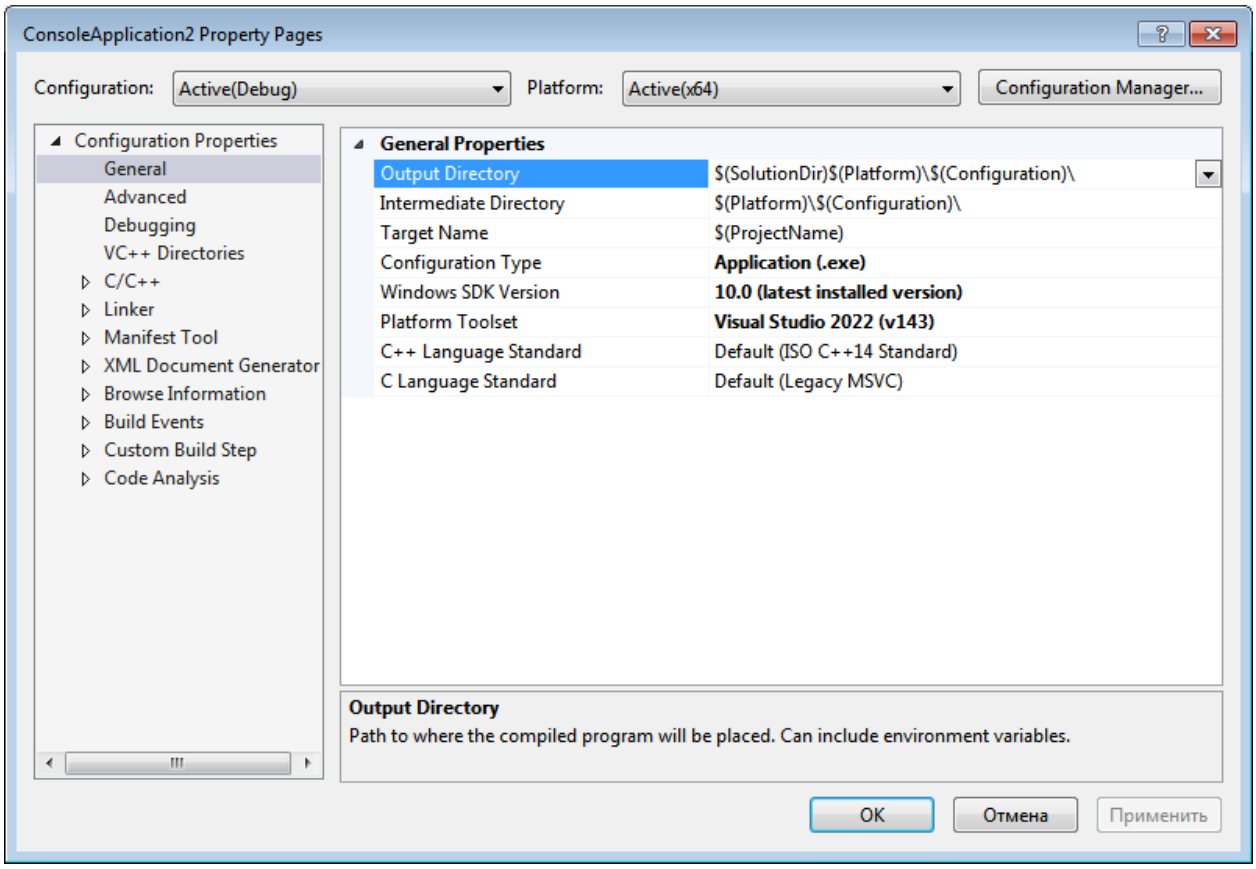


Рис. 7

Розглянути всі властивості проєкту в даній книзі просто нереально (про це можна б було написати окрему книгу). Тому звернемо увагу лише на одну єдину властивість проєкту на вкладці General Properties, яка відкрита за замовчуванням (див. рис. 7). Це передостання властивість C++ Language Standard, яка за замовчуванням встановлена як ISO C++14 Standard (див. праву частину діалогу на рис. 7). За необхідності підтримку стандарту C++ можна встановити як ISO C++17 Standard або ISO C++20 Standard.

Embarcadero Dev-C++

Загальний вигляд вікна IDE з україномовним інтерфейсом показано на

рис. 8 (цей вигляд може дещо відрізнятись залежно від налаштувань IDE). Мову інтерфейсу IDE можна змінити за допомогою команди Tools → Environment Options… або Сервіс → Параметри середовища.

В меню Файл обираємо елемент Створити → Проект…. Відкривається діалог Новий проект (рис. 9). В цьому діалозі обираємо тип проєкту Console Application на активній вкладці Basic, обираємо мову програмування (завжди можна обирати C++) та вводимо потрібне ім'я проєкту (зараз це поле містить Project1).

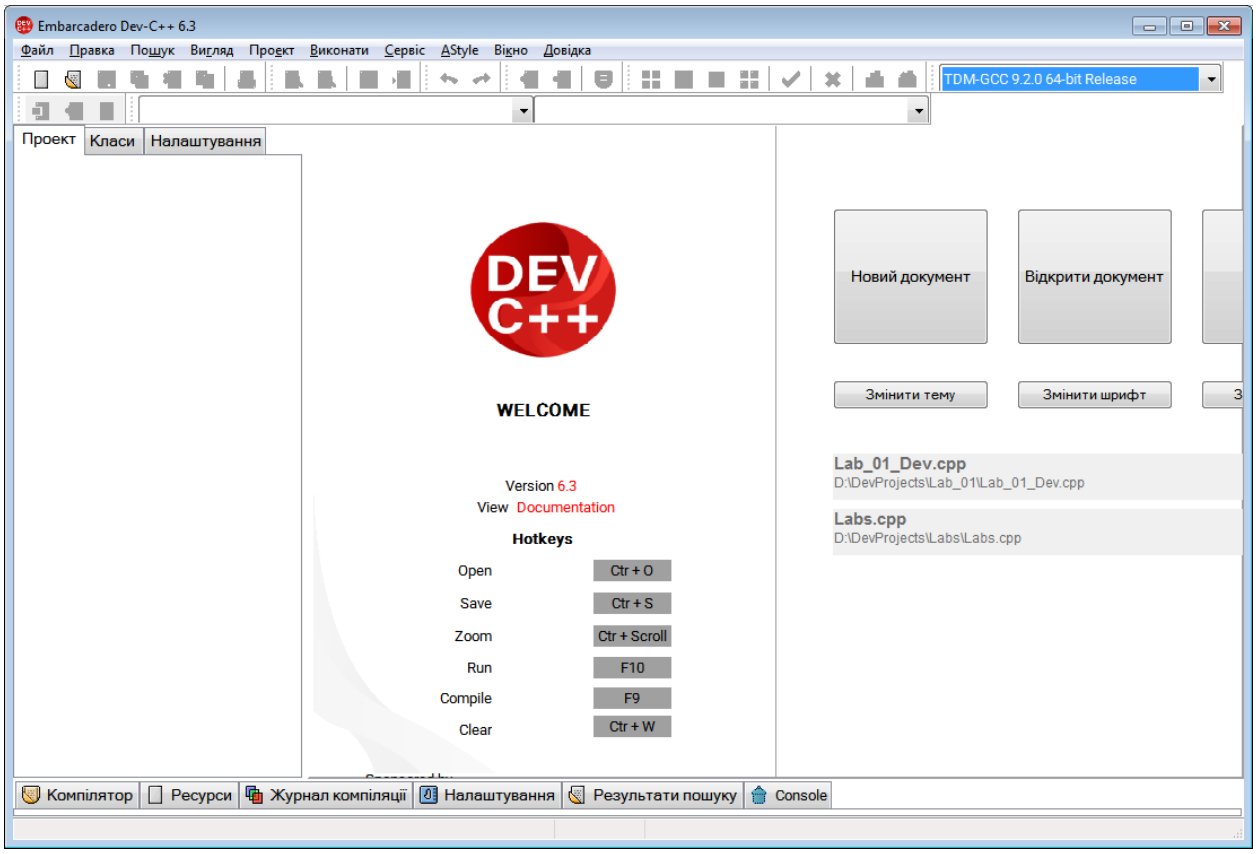


Рис. 8

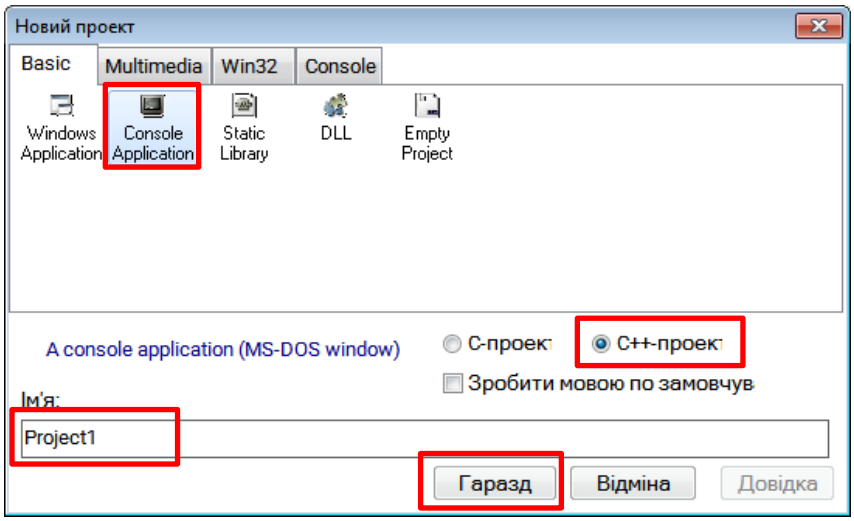


Рис. 9

Після натискання кнопки Гаразд, необхідно вказати місце, де будуть зберігатись файл проєкту з розширенням .dev та .cpp файл(и) з текстом програми. Як тільки ця інформація буде введена, IDE створить проєкт з єдиним .cpp файлом, який має ім’я main.cpp (рис. 10).

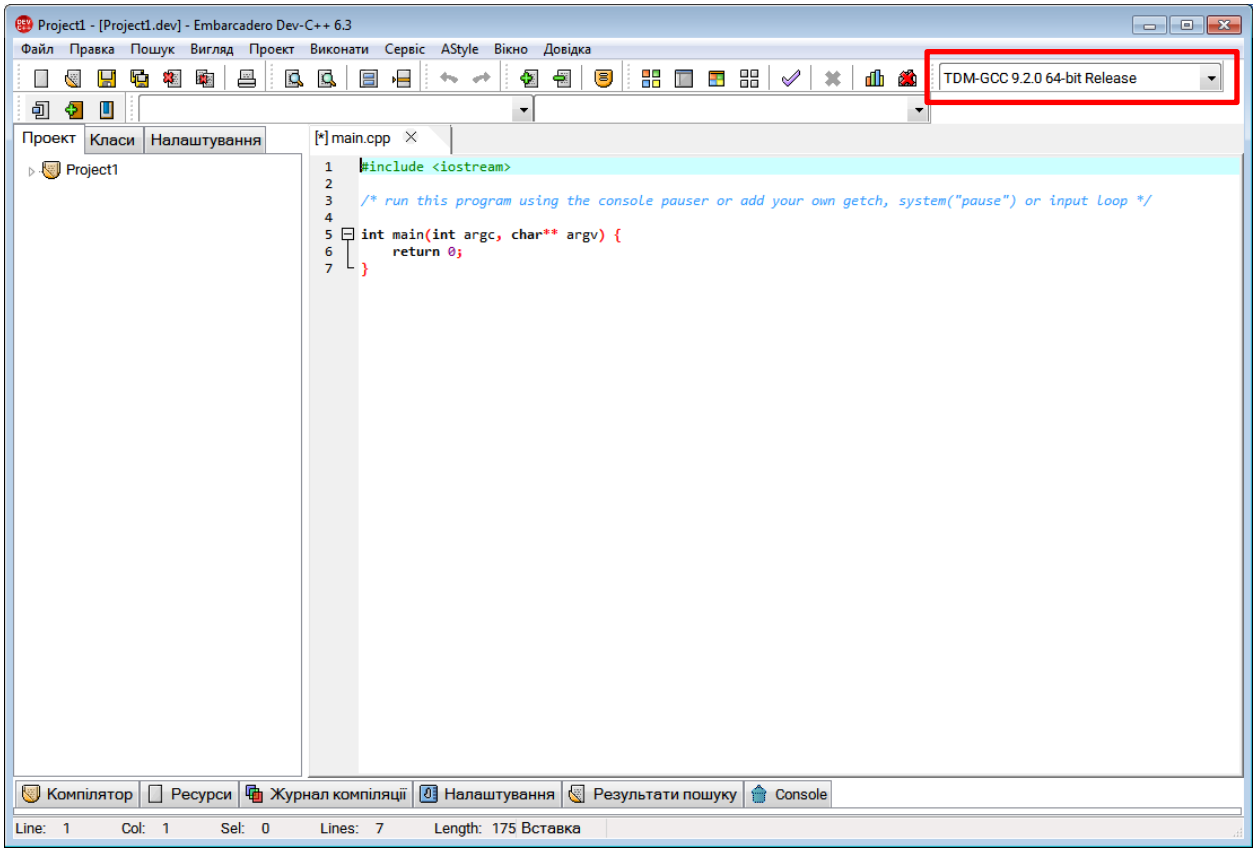


Рис. 10

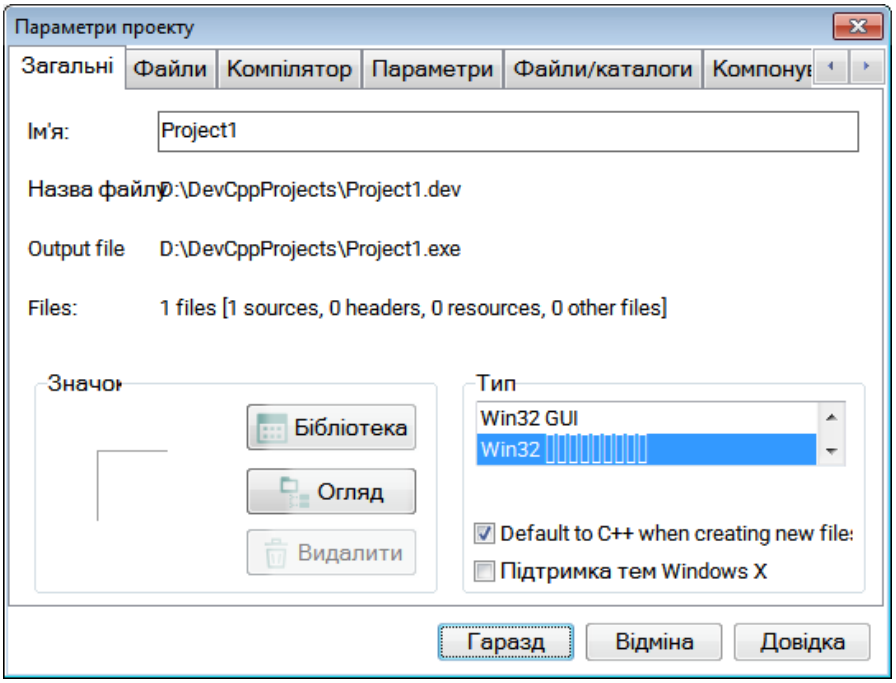


Рис. 11

За замовчуванням Dev-C++ створює фінальну (release) версію програми для 64-розрядних процесів, орієнтовану на досягнення максимальної швидкодії. Цій версії відповідає словосполучення 64-bit Release у випадаючому списку, що задає особливості створення машинного коду програми (рис. 10). За необхідності версію програми можна змінити з фінальної (Release) на налагоджувальну (Debug), також можна змінити розрядність процесора, потрібного для запуску програми (доступні варіанти 32-bit або 64-bit).

Для доступу до властивостей проєкту, обираємо елемент проєкту (з ім'ям Project1) на вкладці Проект і натискаємо праву кнопку миші. У випадаючому контекстному меню обираємо пункт Параметри проекту, що приводить до відкриття діалогу Параметри проекту (рис. 11). В цьому діалозі є декілька вкладок, в яких можна змінювати властивості створеного проєкту.

Code::Blocks

На рис. 12 показано вигляд вікна Code::Blocks після запуску цього IDE (цей вигляд може дещо змінюватись залежно від налаштувань IDE).

Для створення нового проєкту, в меню File вибираємо команду New → Project…. Відкривається діалог New from template (рис. 13), в якому обираємо елемент Console application і натискаємо кнопку Go.

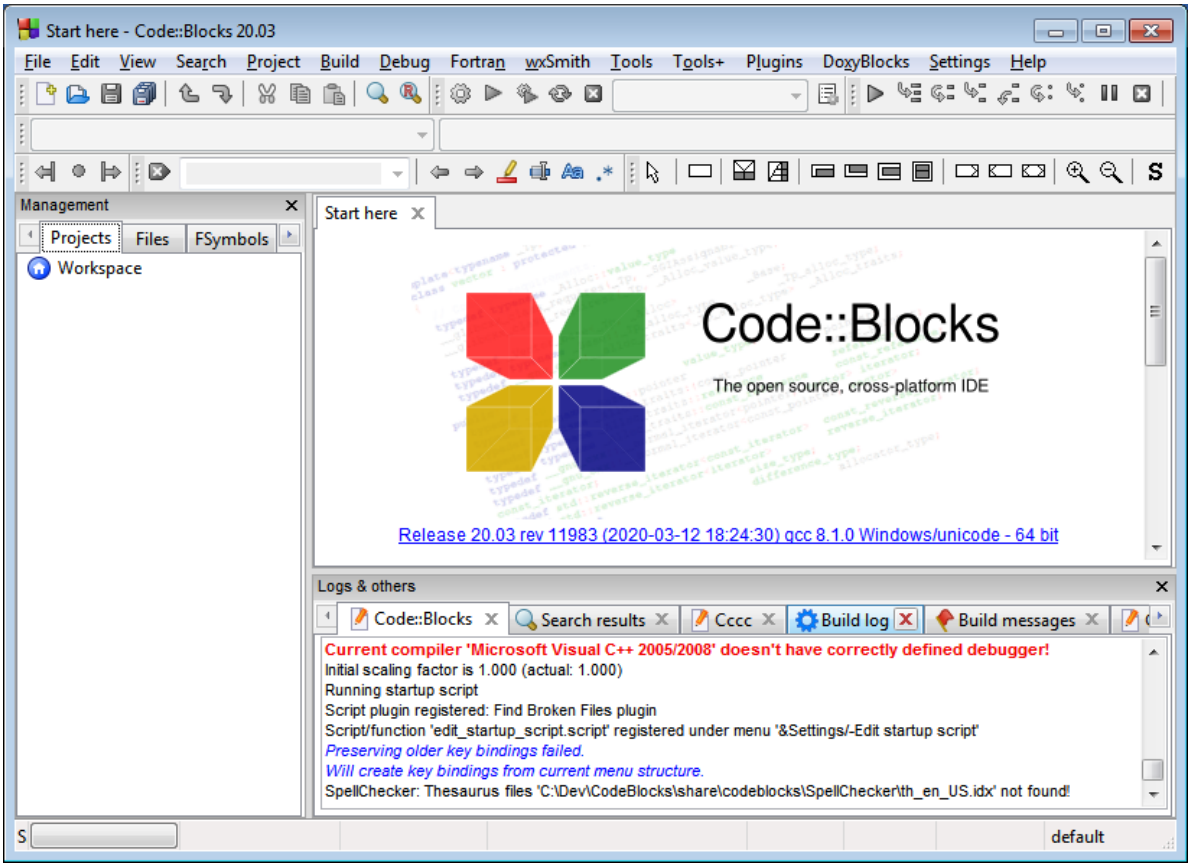


Рис. 12

Запускається майстер створення консольних застосунків. У ньому натискаємо кнопку Next і в новому діалозі, що з'явиться (рис. 14), обираємо бажану мову програмування (достатньо обрати C++).

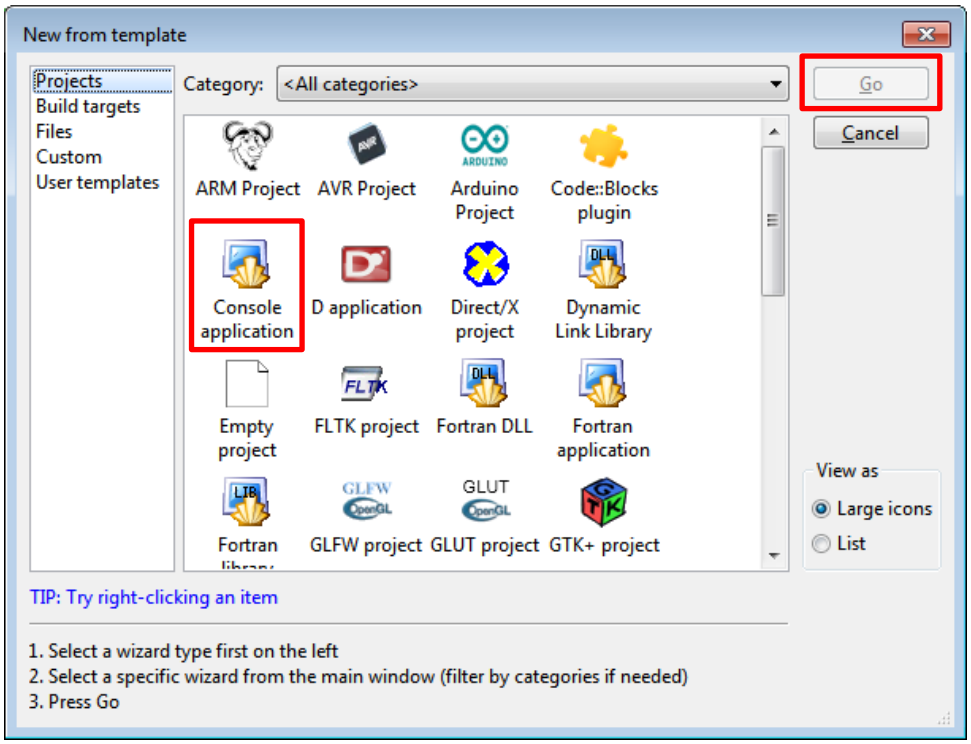


Рис. 13

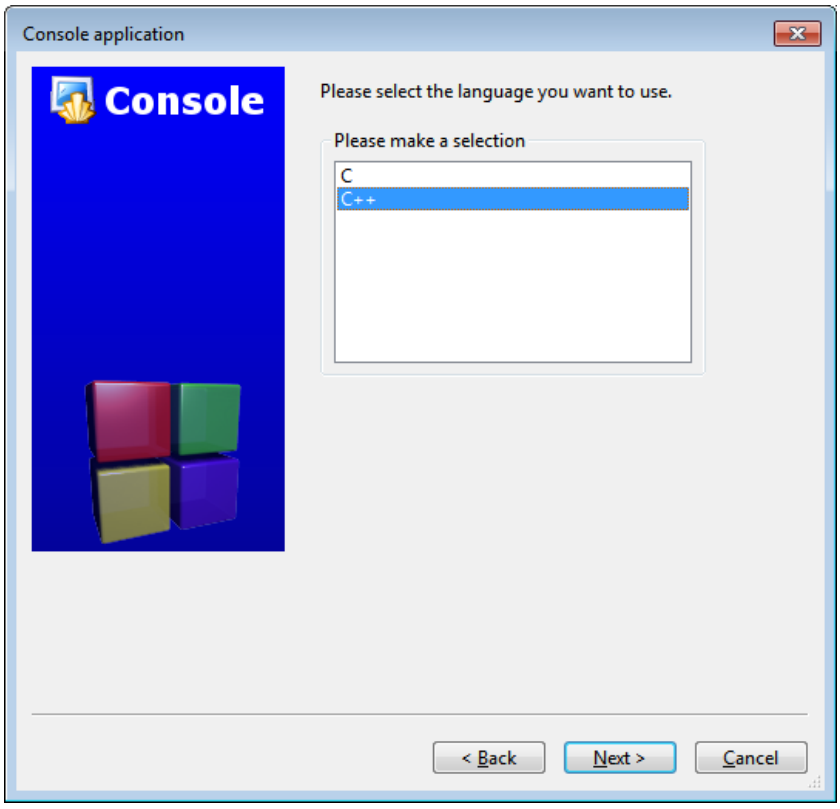


Рис. 14

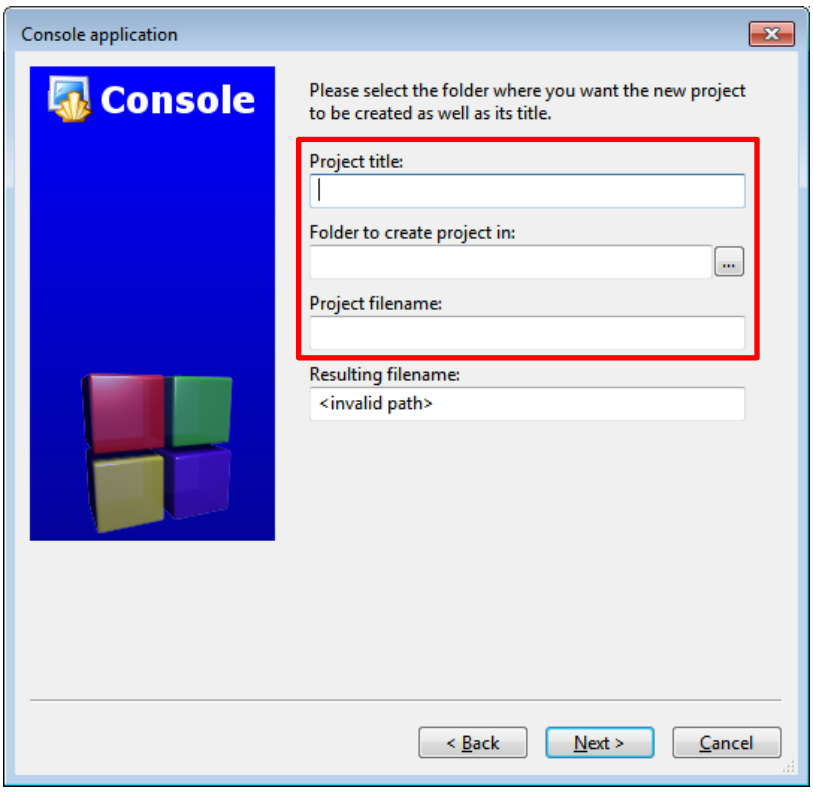

Console application
Console
Please select the folder where you want the new project to be created as well as its title.
Project title:
Folder to create project in:
Project filename:
Resulting filename:
<invalid path>
< Back
Next >
Cancel

Рис. 15

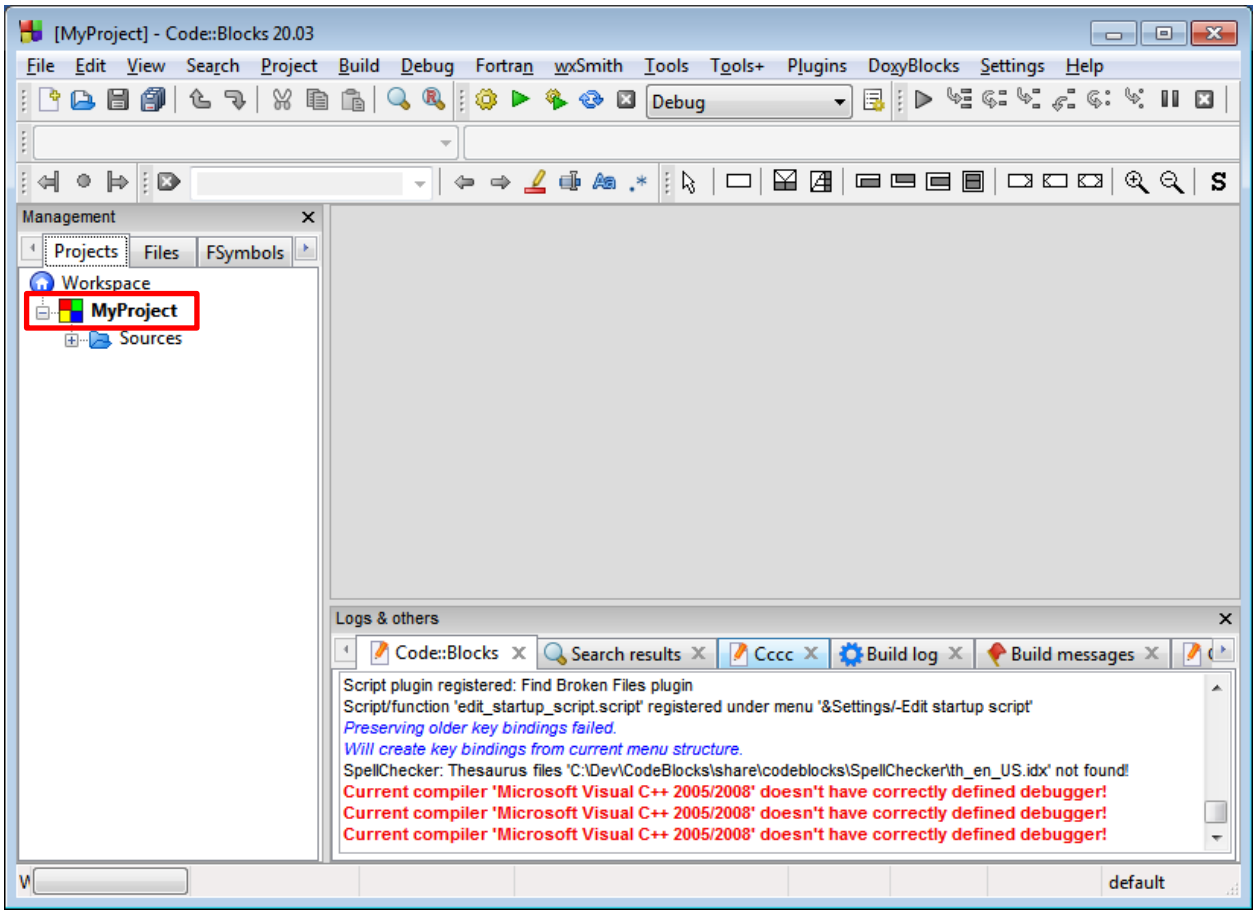

[MyProject] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
Management
Projects Files FSymbols
Workspace
MyProject
Sources
Logs & others
Code::Blocks
Search results
Cccc
Build log
Build messages
Script plugin registered: Find Broken Files plugin
Script/function 'edit_startup_script.script' registered under menu '&Settings/-Edit startup script'
Preserving older key bindings failed.
Will create key bindings from current menu structure.
SpellChecker: Thesaurus files 'C:\Dev\CodeBlocks\share\codeblocks\SpellChecker\th_en_US.idx' not found!
Current compiler 'Microsoft Visual C++ 2005/2008' doesn't have correctly defined debugger!
Current compiler 'Microsoft Visual C++ 2005/2008' doesn't have correctly defined debugger!
Current compiler 'Microsoft Visual C++ 2005/2008' doesn't have correctly defined debugger!
default

Рис. 16

Після цього знову натискаємо кнопку Next і переходимо до задання специфічних рис проєкту (рис. 15): його імені (поле Project title) та папки на диску (поле Folder to create project in), в якій буде створено окрему папку з файлами проєкту. Наприклад, вводимо ім'я проєкту MyProject і шлях до папки D:\CBProjects, після чого натискаємо кнопку Next.

Відкривається ще один діалог, в якому можна обрати версію компілятора, що буде використовуватись, та версії коду програми, які планується створювати (налагоджувальна та/або фінальна). Будемо вважати, що ці параметри змінювати не потрібно, отже, просто натискаємо кнопку Finish, що приводить до створення проєкту.

Вигляд вікна IDE з новоствореним проєктом, який має ім'я MyProject, показано на рис. 16.

Для доступу до властивостей проєкту обираємо другий елемент у вкладці Project (з ім'ям MyProject, нижче вузла Workspace, див. рис. 16) і натискаємо праву копку миші. У контекстному меню, що з'явиться, обираємо самий нижній елемент Properties.... В результаті відкривається діалог Project/targets options (рис. 17), в якому можна подивитись або змінити властивості проєкту.

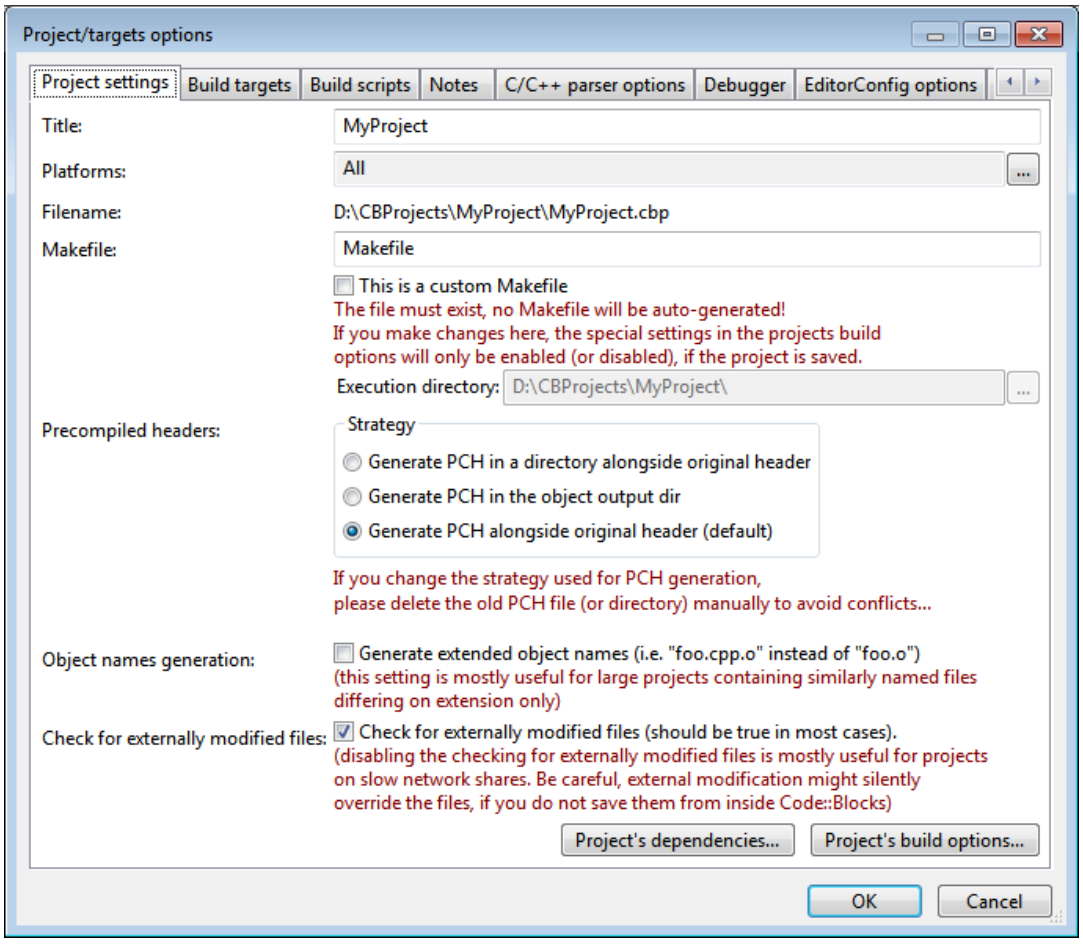


Рис. 17

КОМПІЛЯЦІЯ ТА ЗБИРАННЯ ПРОГРАМ В IDE

В сучасних IDE процеси компіляції та збирання програм часто об'єднують в одну дію – побудову (збирання) всього проєкту (project build). Це виявляється досить зручно, оскільки дозволяє запустити обробку всіх файлів проєкту однією командою. Звичайно ж, за необхідності, програміст може виконувати вибіркову обробку окремих файлів, підключених до проєкту, наприклад, компіляцію лише деяких .c/.cpp файлів.

Процес компіляції та збирання програми C/C++ навіть на сучасних потужних комп'ютерах може тривати декілька секунд (типово 1–60 с). У випадку складних проєктів з великою кількістю файлів, час обробки проєкту може збільшуватись до кількох хвилин (іноді десятків хвилин). Щоб оцінити масштаби роботи, яку проводять препроцесор C/C++, компілятор та компонувальник, також зазначимо, що обробка тексту програми обсягом у кілька рядків (рис. 6), приводить до створення файлу застосунку (66 Кб) та ряду допоміжних файлів із загальним обсягом більше 2 Мб. У складний проєктах обсяг проміжних файлів, які утворюються, може перевищувати 100–1000 Мб.

Оскільки робота з файлами, навіть на сучасних комп'ютерах, виконується відносно повільно, професійні IDE намагаються мінімізувати кількість файлових операцій і за рахунок цього підвищити швидкість обробки проєкту. IDE відслідковують зміни у файлах, що підключені до проєкту. І якщо виявиться, що певний файл не змінювався з моменту його попередньої обробки, перекомпіляція цього файлу не проводиться (відповідний етап пропускається), а замість цього максимально використовуються раніше створені проміжні файли (.i, .o, .obj, .lib тощо). Такий підхід дозволяє суттєво (іноді на порядки) пришвидшити збирання проєкту. Однак у деяких випадках він також може приводити до «ігнорування» змін, внесених програмістом у вихідні файли програми. Для усунення цього недоліку необхідно виконати повну перебудову (перезбирання) проєкту.

Microsoft Visual Studio Community 2022. Для побудови всього проєкту в меню Build слід обрати команду Build Solution або Rebuild Solution. Можна також скористатись командами Build <Ім'я проєкту> або Rebuild <Ім'я проєкту>, де <Ім'я проєкту> – явним чином задане ім'я активного на даний момент проєкту, відкритого в IDE.

Якщо потрібно швидко перевірити певний .c/.cpp файл на наявність синтаксичних помилок, можна спробувати скомпілювати цей файл (за відсутності помилок, компіляція буде успішна). Для цього у меню Build слід обрати команду Compile.

Примусове повне перезбирання проєкту виконується командами Rebuild Solution або Rebuild <Ім'я проєкту>, які викликаються з меню Build.

Embarcadero Dev-C++. Для збирання проєкту використовується команда Скомпілювати в меню Виконати. Для повної перебудови проєкту в тому ж меню Виконати слід обрати команду Перебудувати все.

Code::Blocks. Збирання проєкту в IDE проводиться через команди меню Build. Команди Build та Build workspace з цього меню дозволяють зібрати проєкт, а команди Rebuild та Rebuild workspace – повністю перебудувати його.

СТРУКТУРА ПРОГРАМИ НА C/C++

Текст програми

Текст будь-якої програми мовою C/C++ записується у одному, чи кількох текстових файлах з розширенням .c (для мови програмування C) або .cpp (для мови програмування C++). Часто також використовуються файли заголовків (header – заголовок) з розширенням .h, .hpp або взагалі без розширення[1]. Файли з розширенням .h типово застосовуються при написанні програм будь-якою мовою, C чи C++. Файли заголовків без розширення прийнято застосовувати у програмах на C++.

Код в .c/.cpp файлах складається з ряду характерних блоків, притаманних майже будь-якій програмі мовою C/C++:

1) Пролог програми. В цій секції наводиться різноманітна інформація не пов'язана явним чином зі специфікою коду (алгоритмом роботи) програми, але важлива для створення кінцевого застосунку, налагодження, технічної підтримки програми. Зокрема, в цій секції типово:

а) Наводять коментарі з довідковою інформацією про програму та її автора, призначення певного модуля чи підпрограми тощо;

б) Зазначають бібліотеки підпрограм або окремі стандартні/зовнішні функції, типи даних, які будуть використовуватись в коді програми. Відповідну інформацію також можуть включати до файлу заголовку, який вже потім підключається до проєкту і використовується в тексті програми;

в) Проводять перевірку тих чи інших умов, які мають виконуватись на момент збирання/побудови застосунку. Наприклад, часто перевіряють умови, що: I) застосунок створюється для певної операційної системи; II) використовуються, чи ні в програмі текстові рядки в форматі символів UNICODE; III) виконується чи ні більш надійна перевірка типів даних, що використовуються в програмі тощо;

г) Записують макровизначення (макроси або псевдофункції), які можуть спростити вигляд коду програми. Наприклад, замість того, щоб вводити в програмі декілька разів число π, можна визначити макрос, який пов'язує

[1] При написанні простих, малих за об'ємом програм можна обійтись і без файлів заголовків. На практиці, однак, створювати програму часто виявляється зручніше з цим файлом (або файлами). У файл заголовку, зазвичай, включають найбільш вживані програмні конструкції, які майже не змінюються під час написання програми. Як правило, це визначення типів, специфічних для програми, прототипи функцій, макроси тощо.

текстовий рядок `3.141592653589` з текстовим рядком `PI`, що дозволяє далі замість значення числа π просто писати в програмі `PI`.

2) Визначення типів даних, специфічних для програми. Основна задача будь-яких програм – обробка даних. У мовах C/C++ існує декілька заздалегідь визначених елементарних типів даних для збереження чисел і символів. На основі цих типів даних можна будувати нові, більш складні типи даних. Фактично, будь-яка більш-менш серйозна програма на C/C++ містить специфічні (user defined) типи даних, введені програмістом. У деяких складних проєктах, наприклад, у професійних графічних редакторах, операційних системах, сучасних комп'ютерних іграх, кількість специфічних типів даних, введених програмістами, може сягати кількох тисяч і, навіть, десятків тисяч.

Перш ніж використовувати будь-які специфічні типи даних, їх необхідно визначити. Як правило, це робиться в даній секції програми, ще *до написання* коду програми. Головне щоб тип даних був визначений до того моменту як об'єкт відповідного типу буде використовуватись в тексті програми.

3) Визначення глобальних та статичних змінних, об'єктів даних та констант. Після того як всі потрібні типи даних в програмі визначені (див. п. 2), можна створити довільну кількість глобальних чи статичних змінних, об'єктів чи констант відповідних типів (статичні об'єкти є визначеними тільки в межах даного .c/.cpp файлу). Ці змінні, об'єкти та константи пройдуть процедуру ініціалізації в момент запуску застосунку і будуть доступні протягом усього часу роботи програми.

4) Код програми. Ця секція програми традиційно є найбільшою. Вона включає в себе код для т. зв. точки входу програми – в нашому випадку (для консольних застосунків) це буде функція `main()`. Також у цій секції наводиться код будь-яких інших функцій, написаних програмістом, які використовує програма.

Зробимо ряд зауважень. Сучасні інструменти розробника програм на C/C++ дозволяють програмісту працювати як з дуже малими програмами, так і з проєктами практично необмеженого обсягу, які складаються з тисяч файлів, що містять мільйони рядків коду. Тому наведений вище «план-схема» програми може іноді змінюватись від проєкту до проєкту. Наприклад, типовим підходом до визначення специфічних типів даних програми є запис цих визначень у файлі заголовку з розширенням .h/.hpp або без розширення, а безпосередньо коду – у .c/.cpp файлах. Причому код для роботи з даними певного типу може бути «розмазаним» по кільком .c/.cpp файлам. Однак, як ми побачимо далі при виконанні лабораторних робіт, власне код можна наводити і в файлах заголовків (.h/.hpp або без розширення). Більше того, це є типовим для визначення так званих `inline` функцій, а також `inline` методів класів. Може бути і зворотна ситуація, коли деякий тип даних визначається в .c/.cpp файлі, або в середині іншого типу даних, або, навіть, в середині тіла певної функції.

Тим не менш, розумне слідування наведеній вище схемі побудови тексту програми мовою C/C++ дозволить виробити правильний стиль написання

програм, спростити процедуру їх розробки, відлагодження та можливої подальшої підтримки.

Наведемо тепер приклад найпростішої програми на C/C++:

Лістинг 1:

```
1:  /* Наша перша програма на C/C++! */
2:  #include <stdio.h>
3:
4:  // Функція main() – точка входу в програму.
5:  int main( void )
6:  {
7:      printf("Glory to Ukraine!\n");
8:      return 0;
9:  }
```

Ця програма формально займає всього 9 рядків, з яких значущими є тільки 8 (вільний рядок 3 введено лише для зручності читання/аналізу тексту програми). Цифри ліворуч, наприклад `3:`, позначають номер рядка. В сучасних пакетах розробки програм на C/C++ можна налаштувати відображення номеру рядка, а також використовувати різноманітні кольорові схеми для виділення різних елементів коду програми (див. рис. 18 для коду цієї програми у Microsoft Visual Studio 2022).

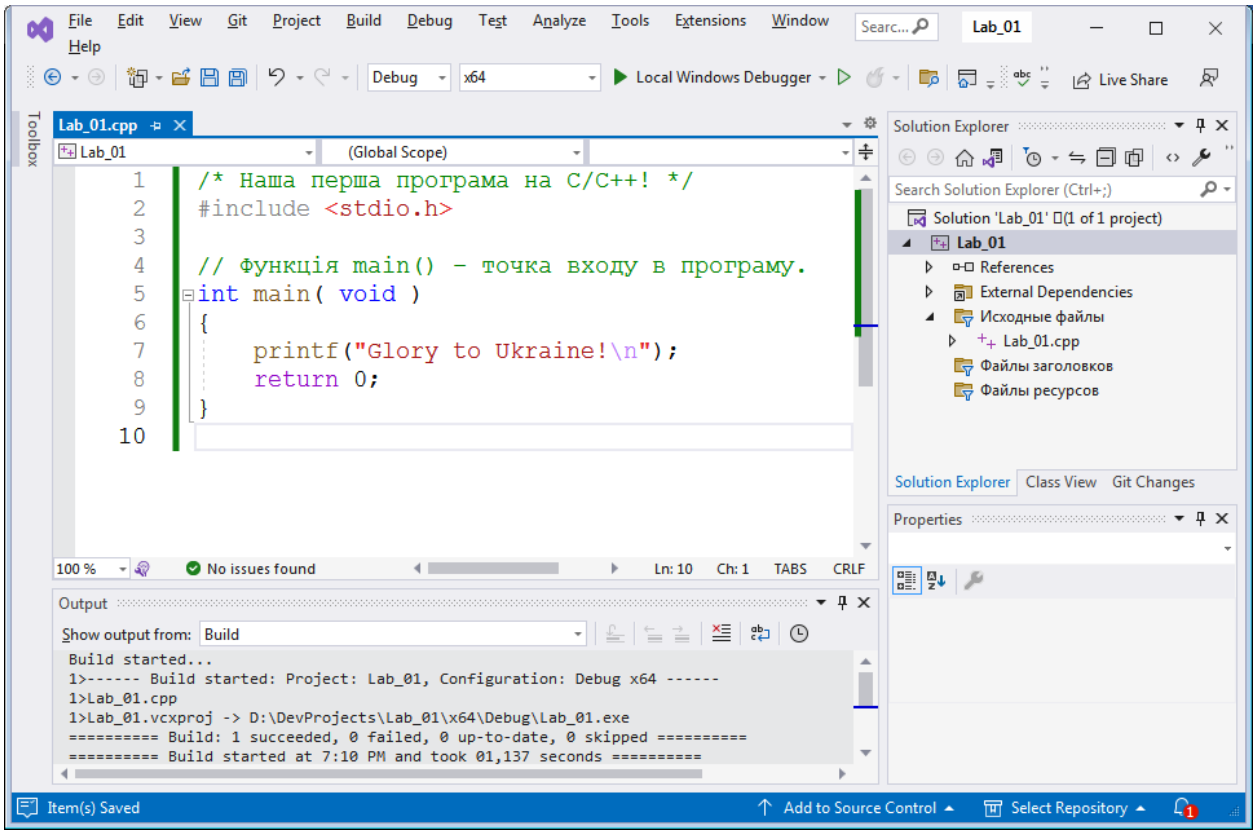


Рис. 18

У лістингу 1 перший та другий рядки є ***прологом програми***. Для наведеної найпростішої програми пролог містить лише коментар в стилі C `/* Наша перша програма на C/C++! */` та звернення до бібліотеки підпрограм `#include <stdio.h>`.

Блоки 2 та 3 в структурі програми відсутні. Визначення специфічних типів даних (блок 2), об'єктів, констант та змінних жодного типу (блок 3) в лістингу 1 не відбувається. Рядок 4 `// Функція main() – точка входу в програму.` є коментарем у стилі C++, він ігнорується при створенні коду програми.

Власне, сам ***код програми*** (блок 4) займає рядки з 5 по 9.

Перш ніж перейти до аналізу коду лістингу 1, зазначимо, що стиль запису тексту цієї програми, наведений вище, не є єдино можливим. Наприклад, та ж сама програма може бути записана як лістинг 2.

Лістинг 2:

```
1: /* Наша перша програма на C/C++! */
2: #include <stdio.h>// Наша перша програма на C/C++!
3: int main(){printf("Glory to Ukraine!\n");return 0;}
```

З точки зору кінцевого результату – створення застосунку – обидві версії програми (лістинги 1 та 2) є абсолютно однаковими. Однак, очевидно, що лістинг 1 є значно більш зручним для аналізу – відповідний код є візуально структурованим, розбитим на блоки, елементам одного рівня відповідають однакові відступи, між ключовими блоками програми є вільний розділовий рядок 3. Крім того, в лістингу 2 аргумент функції `main()` не вказано, що означає за замовчуванням тип `void` – порожній тип (тобто аргументів у функції немає), а в лістингу 1 явно записано `main( void )`.

Розглянемо тепер більш детально лістинг 1.

На початку програми мовою C/C++ (блок 1 – пролог), як правило, записуються директиви препроцесора. Кожна така директива починається зі знаку діеза або решітки `#`. Наприклад, директива `#include` слугує для запису у даний файл (.c/.cpp, .h тощо) вмісту іншого файлу (в лістингу 1 – це файл `stdio.h`). Файл, що включається, формально може бути будь-яким текстовим файлом, але, зазвичай, за допомогою `#include` до тексту програми включають вміст файлів заголовків, наприклад, `#include <math.h>` або `#include <cmath>`.

Використання `#include` дозволяє легко включати вже написані блоки програми до складу різних .c/.cpp/.h та ін. файлів. Найчастіше, директива препроцесора `#include` використовується для включення в текст програми посилання на бібліотеки функцій, типи даних, константи тощо, визначені поза межами поточного .c/.cpp файлу. Ці бібліотеки, функції, типи та інші елементи програми можуть бути як стандартними для мови C/C++ (вбудованими), так і

специфічними, визначеними самим програмістом. Зокрема, рядок

```
#include <stdio.h>
```

підключає до програми файл `stdio.h`, і тим самим дає програмі доступ до стандартних бібліотечних функцій з бібліотеки `stdio` (Standard Input/Output), яка включає в себе функції для вводу-виводу даних на консоль та для роботи з файлами. У лістингу 1 використовується лише єдина стандартна функція `printf()` серед великої сукупності функцій, доступних у бібліотеці `stdio` (див. рядок 7).

Власне програма C/C++ складається із набору підпрограм, кожна з яких може викликати іншу. Кожна така підпрограма називається ***функцією*** (або ***методом***, якщо задана для класів C++). Серед всіх підпрограм одна є головною і є ***точкою входу*** в застосунок – тою функцією, з якої власне і починається виконання коду, написаного розробником програми. Для лістингу 1 ця головна функція має стандартну назву `main()` (див. рядок 5). Саме з неї починається виконання програми, написаної програмістом. З функції `main()` можна викликати інші підпрограми, які в свою чергу, можуть звертатися до наступних підпрограм тощо.

Характерною ознакою функції (методу) є використання круглих дужок `()` після її імені (або імені методу).

Функція `main()` може мати аргументи, вказані у круглих дужках `()`, або ж може викликатися без них. У першому випадку аргументам функції присвоюється значення тих параметрів, які передаються в командному рядку при запуску програми з консолі. В лістингу 1 у дужках функції `main()` вказано `void`, що означає відсутність аргументів (той же самий ефект дає використання пустих дужок `()`). Тобто при такій формі визначення `main()` всі передані їй параметри, фактично, будуть ігноруватися.

Функції можуть не тільки мати аргументи (тобто приймати при своєму виклику відповідні параметри, потрібні для їх роботи), але й можуть повертати результат своєї роботи – значення певного типу. Функція `int main(void)`, записана в лістингу 1, має повертати результат типу `int` – тобто ціле число зі знаком. За допомогою цього коду повернення функція `main()` інформує операційну систему про результат своєї роботи. Традиційно вважають, що нульове значення коду повернення `main()` свідчить про успішне завершення програми, а ненульове значення, як правило, свідчить про помилки у роботі програми.

Тіло функції `main()` обмежене фігурними дужками `{}`. Воно включає в себе два рядки коду (рядки 7 та 8 у лістингу 1). Зазначимо, що окрім виділення тіла функції, символи `{` та `}` можуть використовуватись для виділення довільного блоку коду – т. зв. ***блоку операторів***. На практиці розділові знаки `{}` мови C/C++ вживаються для маркування меж логічних блоків програми,

що містять декілька операторів (команд). Так, наприклад, тіло функції `main()` можна вважати логічним блоком програми, що містить два оператори (рядки 7–8), які в сукупності розв'язують певну єдину задачу.

У рядку 7 викликається стандартна функція `printf()` з аргументом `"Glory to Ukraine!\n"`, яка виводить відповідний текстовий рядок на консоль. Крапка з комою `;` після `printf("Glory to Ukraine!\n")` позначає кінець відповідного оператора (тут – оператора виклику функції).

Рядок 8 програми включає в себе оператор повернення з функції `return`. Форма запису `return 0;` означає, що функція завершується з кодом завершення `0` – саме це значення повертається назовні, тій системній підпрограмі, яка викликала функцію `main()`.

Машинний код програми

Файл машинного коду довільної програми на C/C++ завжди записується у бінарний файл певного формату. Для конкретності розглянемо далі випадок, коли цей файл являє собою застосунок з розширенням .exe (саме такий випадок буде використовуватись в усіх лабораторних роботах).

Файл застосунку .exe має складну структуру, що починається із заголовку, який зчитується в оперативну пам'ять операційною системою під час запуску застосунку (Рис. 19). У заголовку вказується ***точка входу*** в програму на C/C++ – та функція (підпрограма), з якої починається виконання коду, написаного програмістом і специфічного для конкретної програми (для консольних застосунків точкою входу є функція `main()`).

Під час запуску застосунку операційна система:

1) зчитує його машинний (бінарний) код в оперативну пам'ять;

2) виконує код запуску (startup-код) застосунку, під час якого програма отримує доступ до системних ресурсів, виділених ОС для застосунку; в цей момент також виконується ініціалізація внутрішніх ресурсів/даних самої програми, і тільки після цього

3) викликається функція задана як точка входу

Рис. 19

– створений програмістом код програми.

Залежно від вибору цільової операційної системи для застосунку та режиму роботи ОС та/або режиму роботи застосунку, ця функція може мати різне ім’я: `main()`, `WinMain()`, `wWinMain()` тощо.

У програмах для лабораторних робіт як точка входу буде використовуватись функція `main()`, характерна для ***консольних*** застосунків. Консольні застосунки є найпростішими. Вони, фактично, не мають повноцінного графічного інтерфейсу – за замовчуванням їх графічним інтерфейсом керує не сам застосунок, а операційна система. Робота з консольним застосунком нагадує роботу із застосунками старих операційних систем типу MS-DOS. У консольному застосунку для введення/виведення даних використовується псевдографічна консоль, яка емулює текстовий режим введення/виведення інформації (див. рис. 20, де показано приклад вікна стандартного консольного застосунку cmd операційної системи Windows 7).

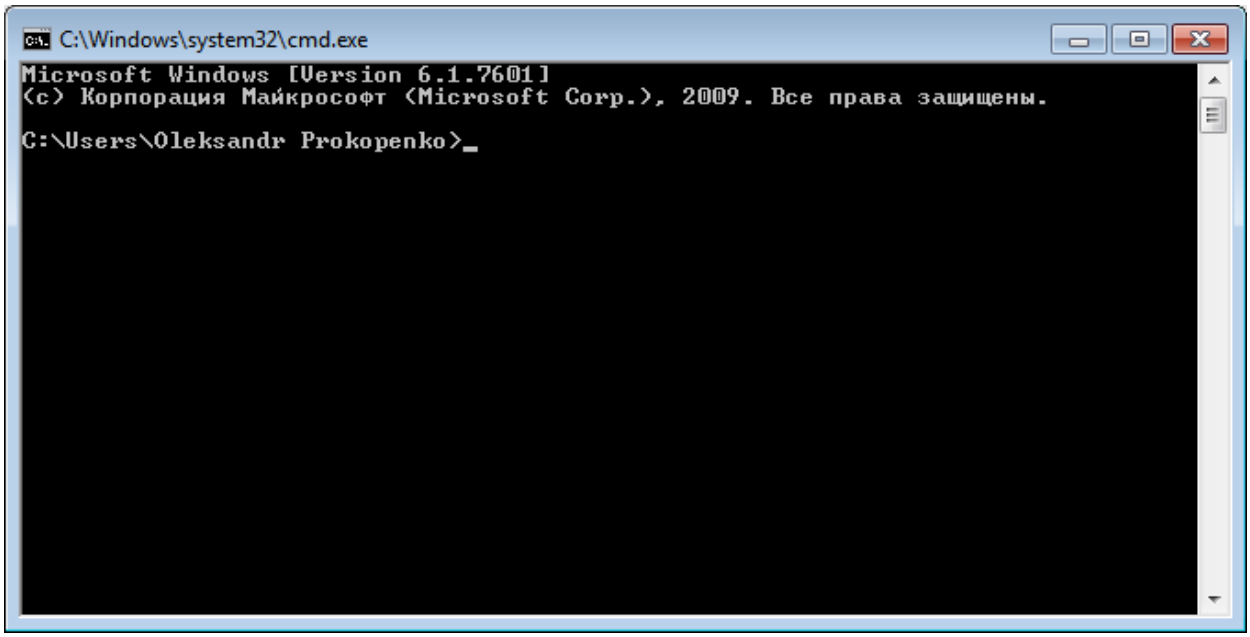


Рис. 20

Лабораторна робота № 1.
Найпростіші операції з даними.
Введення/виведення даних на консоль

Мета роботи:

1) Набути практичні навички роботи з інтегрованим середовищем розробки програм мовою C/C++, компілятором та компонувальником C/C++.
2) Навчитися працювати зі змінними інтегральних типів даних (`int`, `char`, `float`, `double` тощо) та константами відповідних типів.
3) Опанувати використання найпростіших операцій з математичними та символьними даними у програмі на C/C++.
4) Навчитися зчитувати дані з командного рядка (консолі) та виводити дані на консоль.

ПРОГРАМА ЯК НАБІР ФУНКЦІЙ ТА ОПЕРАТОРІВ

Як вже зазначалось раніше (див. «Структура програми на C/C++»), текст програми мовою C/C++ може включати в себе кілька характерних структурних блоків. Один з цих блоків, т. зв. ***блок коду*** (див. рядки 5-9 лістингу 1), власне, і містить *текст програми*, який буде перетворюватись у *машинний код* (застосунку, бібліотеки тощо).

Для консольних проєктів, які будуть використовуватись у цьому практикумі, блок коду обов'язково повинен містити функцію `main()`. Він також може включати й інші функції (див. лабораторну роботу 3).

Код програми, пов'язаний з функцією, зокрема, з функцією `main()` – див. лістинг 1, записується у фігурних дужках `{}`. Цей код називається ***тілом функції*** і являє собою послідовність операторів.

Оператор – спеціальна конструкція мови C/C++, яка керує процесом виконання програми. Оператор можна умовно ототожнити з певною дією, яка має виконуватись в програмі. Таким чином, будь-яка програма на C/C++ може розглядатись як набір операторів, що виконуються в певній послідовності. Ці оператори можуть бути записані як в тілі однієї функції (у нашому випадку – функції `main()`), так і можуть бути присутніми у функції `main()` та багатьох інших функціях.

Оператори бувають прості та складні.

Прості оператори виконують певну достатньо просту дію і в тексті програми обов'язково мають завершуватись крапкою з комою `;` (див. рядки 7-8 лістингу 1). Наприклад, код у рядку 7 використовує ***оператор виклику функції*** `()`, що застосовується до функції з ім'ям `printf`, якій передається

єдиний аргумент – текстовий рядок `"Glory to Ukraine!\n"`. У рядку 8 використовується ***оператор повернення*** `return`, який повертає значення 0 startup-коду, який викликав функцію `main()`.

Складні оператори або ***блоки операторів*** – це сукупність простих операторів, включених у фігурні дужки `{}`. Фігурна дужка, що відкривається, `{`, позначає початок складного оператора, а фігурна дужка, що закривається, `}`, позначає кінець відповідного блоку операторів. Всередині блоку операторів можна записати довільну кількість простих або складних операторів. Тобто складні оператори можуть містити вкладені блоки операторів (вкладені складні оператори), які, в свою чергу, можуть містити ще вкладені блоки операторів, тощо. Все це разом формує унікальну структуру тексту програми як послідовності операторів.

Наведемо приклади запису простих і складних операторів:

Лістинг 1.1:

```
 1: { int i = 5; }
 2: {
 3:     int i = 5;
 4:     {
 5:         int j = i + 1;
 6:     }
 7: }
 8: {
 9:     int i = 5;
10:     {
11:         int j = i + 1;
12:         {
13:             int k = j + 1;
14:         }
15:     }
16: }
```

Код у рядку 1 з лістингу 1.1 являє собою простий оператор `int i = 5;`, який формально записано у фігурних дужках `{}` як складний оператор. Якщо прибрати ці фігурні дужки, зміст програми не зміниться.

Код у рядках 2-7 є складним оператором, що містить один простий оператор `int i = 5;` (рядок 3) та вкладений складний оператор у рядках 4-6.

Нарешті, складний оператор, записаний у рядках 8-16, містить вкладений блок операторів (рядки 10-15), всередині якого є ще один вкладений блок операторів (рядки 12-14).

Якщо розглядати код програми як послідовність операторів, тіло функції, обмежене фігурними дужками `{}` (наприклад, тіло функції `main()` у лістингу 1), можна вважати блоком операторів. Особливістю такого блоку є те,

що, на відміну від звичайного блоків операторів з лістингу 1.1, він має «власне унікальне ім'я» – ім'я функції.

ЗМІННІ, КОНСТАНТИ ТА ТИПИ ДАНИХ

Програма здійснює операції з даними. Для збереження цих даних використовуються змінні. Під ***змінною*** розуміється деяка область пам'яті, якій присвоєне власне ім'я, і дані якої можуть бути зчитані або записані.

Різні ***типи даних*** вимагають різного об'єму відділеної пам'яті та, можливо, різної організації даних у ній. Тому перед використанням змінної її треба ***оголосити***, вказавши тип даних змінної та її унікальне ім'я – ***ідентифікатор*** змінної. Це робиться, наприклад, наступним чином:

```
/* Оголошення змінних: */
int i;   // Змінна і цілого типу (int).
float x; // Змінна х дійсного типу одинарної
         // точності (float).
char a;  // Змінна а символьного типу (char).
```

Під час оголошення змінної можна явно вказати її початкове значення. Також для економії місця змінні одного типу можна оголошувати в одному рядку, записуючи їх через кому:

```
int i,j=10,k=500;
float x=3.14, y;
```

Змінні, які оголошуються поза межами яких-небудь функцій називаються ***глобальними***. Вони доступні для використання в довільному місці програми (тобто можна зчитати їх значення або присвоїти їм нове значення в довільному місці програми). Трохи схожими на глобальні є ***статичні*** змінні, які оголошуються з додатковим службовим словом `static`, наприклад,

```
static int s = 100;
```

Статичні змінні доступні для використання лише в межах даного модуля – файлу .c/.cpp.

Змінні, які оголошуються всередині функції або підпрограми називаються ***локальними*** – їх можна використовувати лише всередині відповідної функції, відповідного блоку операторів. Спроба звернутися до таких локальних змінних з іншої функції або іншого блоку призведе до помилки. З іншого боку, це дозволяє без проблем використовувати в різних підпрограмах (функціях) або блоках коду змінні з однаковими іменами. Наприклад, якщо в функції

`f1()` використовується змінна з ім'ям `i`, змінну з таким же ім'ям можна вільно використовувати й в інших функціях. Наприклад, змінні `i`, `j`, `k` з лістингу 1.1 є локальними. Вони визначені лише в межах того блоку, обмеженого дужками `{}`, в якому вони записані. Тобто змінна `k` доступна лише в рядку 13, а змінна `j`, введена в рядку 11, може використовуватись лише в рядках 11-14.

У мові C/C++ є декілька простих (вбудованих, базових або інтегральних) ***типів даних***: `int`, `float`, `double`, `char` тощо, інформація про які наведена в таблиці:

Тип	**Які дані можуть зберігатись у змінній відповідного типу**	**Об'єм пам'яті**	**Характерний діапазон значень**
`int`	Цілі числа (32-бітні)	4 байти	від –2147483648 до +2147483647
`float`	Дійсні числа одинарної точності	4 байти	від $\pm 3.4\times10^{-38}$ до $\pm 3.4\times10^{38}$ (приблизно)
`double`	Дійсні числа подвійної точності	8 байт	від $\pm 1.7\times10^{-308}$ до $\pm 1.7\times 10^{308}$ (приблизно)
`char`	Символи кодової таблиці ASCII (фактично – коди символів)	1 байт	–128 … +127
`bool` (мова C++)	Булевий (логічний) тип	1 байт	`false`, `true`

Базові типи `int` та `char` можуть використовуватися разом з додатковими модифікаторами `unsigned` або `signed`. Крім того, тип `int` – з ще одним додатковим модифікатором `short` або `long`, а тип `double` з додатковим модифікатором `long`.

Модифікатор типу `unsigned` (без знаку) вказує на те, що змінна може зберігати тільки значення без знаку. Наприклад, тип `unsigned char` дозволяє зберігати тільки значення від 0 до 255. Модифікатор типу `signed` (зі знаком) вказує на те, що змінна відповідного типу завжди має використовуватись як величина зі знаком. Модифікатор `signed` для типу `int` передбачається за замовчуванням, тому, наприклад, типи `signed int` та `int` є еквівалентними. Використання модифікаторів `signed` та `unsigned` для типу `char` має певні особливості, зокрема, типи `char`, `signed char` та `unsigned char` можуть розглядатись як три різні типи.

Змінні цілого типу `int` можуть мати додаткові модифікатори `long` та `short`. Модифікатор `long` *може* збільшувати об'єм пам'яті, який виділяється на збереження змінної, а модифікатор `short` – навпаки, *може* скорочувати цей об'єм пам'яті. Ефект від використання цих модифікаторів залежить

від стандарту C/C++, версії компілятора та операційної системи тощо. Якщо скористатись оператором мови C/C++ `sizeof`, який повертає кількість байтів, необхідних для збереження змінної/об'єкту певного типу, то для модифікаторів `long` та `short` мають виконуватися співвідношення:

$$\texttt{sizeof(long int)} \geq \texttt{sizeof(int)} \geq \texttt{sizeof(short int)}.$$

Зокрема, в програмах, написаних для 32-бітних операційних систем, типи `long int` та `int` займають однаковий об'єм пам'яті – 4 байти, а тип `short int` – 2 байти. У 16-бітних застосунках `sizeof(long int)` = 4 байти, а типи даних `short int` та `int` є еквівалентними і займають 2 байти.

При використанні модифікаторів `long` та `short` базовий тип `int` можна не вказувати, тобто замість `short int` та `long int` можна писати просто `short` та `long`.

Тип `long double` має об'єм 10 байт (розмір регістра математичного співпроцесора) і використовується для збереження дійсних чисел розширеної точності. Проте в більшості програм на C/C++ цей тип неявно інтерпретується як тип `double`. Якщо спеціально не перевести співпроцесор та програму мовою C/C++ в режим роботи з числами розширеної точності, буде вважатися, що `long double` – це тип еквівалентний типу `double` (наприклад, це характерно для операційних систем Windows). У більшості програм/ОС оператор `sizeof(long double)` дає результат 8 (байт), тобто той же результат, що й для типу `double`.

Нарешті, тип `void` використовується при формальному описі функції, яка не повертає ніякого значення або не має аргументів. Також тип `void` застосовується для опису вказівників довільного типу (це питання буде обговорюватись в наступних лабораторних роботах).

Окрім звичайних змінних, у програмах на C/C++ можуть використовуватись константи. ***Константи*** – це іменовані області пам'яті, дані яких після ініціалізації не можна змінювати. Подібно змінним, значення констант можна зчитувати довільну кількість разів, але задати значення константи можна лише один раз, під час запуску програми. Після такої ініціалізації, протягом всього наступного часу роботи програми значення константи не повинно змінюватись. Для створення констант перед її типом пишеться ключове слово `const`, а після її ідентифікатора та знаку `=` наводиться значення константи (все робиться так само як ініціалізація змінних). Ось приклади запису констант:

```
const int    LuckyNumber = 3;
const double MathPI      = 3.1415926;
const char   Symbol      = 'A';
const char*  strText     = "Hello!";
```

У першому прикладі визначається константа типу `int` з ім'ям

`LuckyNumber`, яка ініціалізується значенням `3`. У другому прикладі створюється константа `MathPI` дійсного типу, якій присвоюється дійсне число `3.1415926`. Для ініціалізації константи символьного типу `char` використовується символ `A`, який згідно правил C/C++ має включатись в одинарні лапки `'`, що дає `'A'`. Нарешті, текстовий рядок `"Hello!"`, обмежений подвійними лапками `"` використовується для ініціалізації текстової (рядкової) константи з ім'ям `strText` і типом `char*` (більш детально текстові рядки будуть розглянуті в лабораторній роботі 4).

НАЙПРОСТІШІ ОПЕРАЦІЇ З ДАНИМИ

Аналогом математичних формул у програмуванні є вирази, що складаються з операцій (математичних, логічних, побітових тощо) та операндів (величин, над якими відбуваються ці операції). Операції діляться на ***унарні*** (один операнд), ***бінарні*** (два операнди) та ***тернарні*** (три операнди). Деякі унарні та бінарні операції, що застосовуються в C/C++ перераховані в наступній таблиці:

Позначення	Зміст	Приклад коду	Результат
Деякі унарні операції C/C++			
`+`	Унарний плюс	`int i = 1;` `int j = +i;`	j = +1
`-`	Унарний мінус	`int i = 1;` `int j = -i;`	j = –1
`++`	Інкремент (збільшення значення на 1)	`int i = 1;` `int j = ++i;`	i = 2, j = 2
`--`	Декремент (зменшення значення на 1)	`int i = 1;` `int j = i--;`	i = 0, j = 1
`sizeof`	Визначення розміру в байтах	`int i = 1;` `int j = sizeof(i);`	j = 4
`(тип)`	Перетворення типу	`int i = 1;` `float f = (float)i;`	f = 1.0
Деякі бінарні операції C/C++			
`=`	Присвоєння	`int i = 1;`	i = 1
`+`	Додавання	`int i = 1;` `int j = i + 5;`	j = 6
`–`	Віднімання	`int i = 1;` `int j = i - 3;`	j = –2

Позначення	Зміст	Приклад коду	Результат
*	Добуток	`int i = 2;` `int j = i * i;`	j = 4
/	Ділення	`int i = 7;` `int j = i / 2;`	j = 3
%	Залишок від ділення	`int i = 7;` `int j = i % 2;`	j = 1

Окрім базової форми оператора присвоєння в програмах мовою C/C++ можуть використовуватись скорочені форми запису, показані в таблиці:

Звичайна форма запису	Скорочена
`A = A + B`	`A += B`
`A = A - B`	`A -= B`
`A = A * B`	`A *= B`
`A = A / B`	`A /= B`
`A = A % B`	`A %= B`

Однією з найбільш цікавих операцій, яка часто використовується в програмах мовою C/C++ є ***операція перетворення*** (***приведення*** або ***конвертації***) ***типу***. Ця операція дозволяє змінити інтерпретацію вмісту певного об'єкта в пам'яті. Припустимо, що деякий об'єкт в пам'яті займає 4 байти і являє собою певну послідовність бітів, встановлених у значення «0» або «1». Ця бітова послідовність може кодувати ціле число без знаку типу `unsigned int`, ціле число зі знаком типу `int`, дійсне число типу `float` і навіть послідовність чотирьох символів типу `char`. В усіх цих випадках, зчитуючи одну і ту ж саму бітову послідовність, одержимо суттєво різні дані завдяки різному алгоритму інтерпретації вказаного набору бітів.

У мові C/C++ є різні форми запису операції перетворення типу. Найбільш класичною формою запису є форма `(Тип)Вираз` або `Тип(Вираз)`, яка дозволяє розглядати `Вираз` як величину вказаного `Типу`. Ця форма запису була введена ще в мові C. Більш нові форми перетворення типу, характерні для мови C++, мають вигляд `static_cast<Тип>(Вираз)`, `dynamic_cast<Тип>(Вираз)` та `reinterpret_cast<Тип>(Вираз)`. Ці оператори перетворення типу будуть розглядатись пізніше. Поки що ж лише зазначимо, що оператор C++ для перетворення типу `static_cast<Тип>(Вираз)` працює ідентично спрощеному оператору приведення типу `(Тип)Вираз` або `Тип(Вираз)`.

У деяких випадках, аналізуючи текст програми, компілятор може реалізовувати операцію перетворення типу автоматично, навіть, коли оператор перетворення типу відсутній. Це відбувається всюди, де необхідно виконати т. зв. ***розширення типу*** – тобто перейти від «меншого» типу даних до

«більшого»[1]. Наприклад, від типу `char` (об'єм 1 байт) таким чином можна перейти до типу `short` (2 байти), потім до типу `int` (4 байти) і, нарешті, до типу `double` (8 байт). Якщо ж має відбутись перехід від «більшого» типу до «меншого» – т. зв. ***звуження типу***, це потребує явного використання оператора приведення типу.

Більш складні математичні операції, наприклад, обчислення тригонометричних, показникових чи логарифмічних функцій, модуля чи квадратного кореня числа тощо, реалізуються за допомогою стандартних математичних функцій, реалізованих в стандартній бібліотеці C/C++. Доступ до цієї бібліотеки можна отримати, підключивши до програми файл заголовку `math.h` за допомогою директиви препроцесора `#include <math.h>` (такий підхід характерний для програм C), або використавши директиву `#include <cmath>` для підключення файлу заголовку `cmath` (типово для C++).

Введення/виведення даних за допомогою консолі

Робота з консоллю за допомогою функцій бібліотеки C

Форматований вивід даних на консоль (екран) в програмах на мові C, зазвичай, реалізується за допомогою стандартної функції `printf()` з бібліотеки функцій вводу-виводу `stdio` (Standard Input/Output). Синтаксис виклику цієї функції наступний (наприклад, див. рядок 7 у лістингу 1):

```
printf("Командний рядок", Арг1, Арг2, ...);
```

Командний рядок може містити звичайний текст, який буде виводитися на консоль без змін, спеціальні команди форматування – ***керуючі символи*** або т. зв. ***escape-послідовності*** `\n` – новий рядок, `\r` – повернення каретки, `\t` – горизонтальна табуляція тощо, та/або специфікації форматування для аргументів `Арг1`, `Арг2`, ..., які йдуть після командного рядка. Кожна специфікація застосовується до відповідного аргументу по черзі. Специфікація починається із символу `%`, і закінчується специфікатором типу даних (див. нижче). У проміжку між ними можуть (однак це не є обов'язковим) стояти мінімальна ширина поля для виводу значення змінної, специфікація точності у форматі `X.Y` (мінімальна кількість символів `X` для змінних типу `int` або кількість символів `Y` після крапки для типів `double` чи `float`) тощо.

Декілька найпоширеніших специфікаторів типу даних наведено нижче:

[1] Епітети «менший» та «більший» тип тут означають менший та більший діапазони зміни даних для вказаних типів, або менший та більший об'єм пам'яті, що виділяється на зберігання даних відповідного типу.

%d або %i – ціле число (десяткова система числення);
%f або %g – дійсне число з плаваючою крапкою;
%e або %E – дійсне число в експоненціальній формі запису;
%c – символ, код якого указано в аргументі функції printf();
%s – текстовий рядок (рядок символів);
%x або %X – ціле число у шістнадцятковій формі (по основі 16);
%p – виведення вказівника.

Приклад використання функції printf():

```
printf("Pi = %.4f \n",3.1415926);
```

У цьому прикладі функція printf() виводить текстовий рядок Pi = і потім число, вказане в аргументі, з точністю до 4 знаків після крапки. Результат:

```
Pi = 3.1416
```

Для вводу значень змінних з клавіатури в консольному режимі може використовуватися функція scanf() або її розширена версія scanf_s() з синтаксисом:

```
scanf("Командний рядок", Арг1, Арг2, ...);
scanf_s("Командний рядок", Арг1, Арг2, ...);
```

Функції scanf(), scanf_s() працюють подібно до функції printf(), але командний рядок для них повинен складатися тільки із специфікацій форматування. Аргументами функції scanf(), scanf_s() є *адреси* змінних відповідних типів, яким будуть присвоєні введені з клавіатури дані. На те, що це саме адреси, а не власне змінні, вказують символ & перед іменами змінних, наприклад, команда

```
scanf("%d%f", &x, &y);
```

зчитує з консолі два числа, інтерпретує їх як ціле (специфікатор %d) та дійсне (специфікатор %f), відповідно, і присвоює їх значення змінним x та y. При цьому введення з клавіатури символів, що відповідають різним змінним, потрібно відділяти натисканням клавіші Enter.

Приклад програми, що використовує printf() та scanf(), наведено у лістингу 1.2. У рядку 5 в цій програмі оголошується змінна i цілого типу. У рядку 6 виводиться прохання до користувача ввести ціле число, яке зчитується scanf() у змінну i (див. рядок 7). Введене значення виводиться за допомогою функції printf(), яка викликається в рядку 8.

Лістинг 1.2:

```
 1: #include <stdio.h>
 2:
 3: int main( void )
 4: {
 5:     int i;
 6:     printf("Please input integer number: ");
 7:     scanf("%d", &i);
 8:     printf("You have entered number %d\n", i);
 9:     return 0;
10: }
```

Використання функції `scanf()` є зручним, але може приводити до помилок і збоїв програми, якщо користувач вводить невірні дані (які не відповідають потрібному типу). Для створення більш надійних програм зручніше застосовувати комбінацію функцій `gets()` або функцій, подібних до неї, та `atoi()`, `atof()`. Функція `gets()` зчитує рядок введених символів у текстовий буфер певного розміру. А функції `atoi()` та `atof()` конвертують вміст цього текстового буферу в ціле або дійсне число. Ось приклад програми, що використовує функції `gets_s()`[1] та `atoi()`:

Лістинг 1.3:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>     // Потрібно для atoi().
 3:
 4: int main(void)
 5: {
 6:     char buffer[80];
 7:     printf("Please input integer number: ");
 8:     gets_s(buffer, sizeof(buffer)/sizeof(char));
 9:     int i = atoi(buffer);
10:     printf("You have entered number %d\n", i);
11:     return 0;
12: }
```

Розглянемо особливості цієї програми. У рядку 6 оголошується змінна `buffer` – масив з 80 елементів типу `char` (більш детально масиви будуть розглядатись в наступних лабораторних роботах). У рядку 7 виводиться прохання до користувача ввести ціле число. Коли користувач вводить число і натискає Enter, результат – рядок символів – зчитується функцією `gets_s()` у змінну `buffer` (рядок 8). Ця функція є розширеною (більш безпечною)

[1] Функції із закінченням `_s` у своєму імені є розширеними (більш безпечними) варіантами функцій зі стандартної бібліотеки C.

реалізацією стандартної функції `gets()`. Першим параметром при виклику `gets_s()` передається ім'я буфера даних `buffer`, а другим параметром – розмір буфера в символах `sizeof(buffer)/sizeof(char)` (тут `sizeof(buffer)` – розмір всього буфера, а `sizeof(char)` – розмір одного символу типу `char`). Потім, у рядку 9, викликаючи функцію `atoi()`, відбувається конвертація масиву символів у ціле число, значення якого зберігається в змінній `i`. Нарешті, результат перетворення (значення `i`) виводиться на консоль за допомогою функції `printf()` (рядок 10).

Робота з консоллю за допомогою потоків C++

У програмах мовою C++ для роботи з консоллю використовуються ***потоки введення-виведення***. За замовчуванням для використання доступні чотири стандартні об'єкти потоків: `cin`, `cout`, `cerr` та `clog`:

Об'єкт потоку	Призначення за замовчуванням
`cin`	Введення даних з клавіатури
`cout`	Виведення текстової інформації на консоль
`cerr`	Виведення інформації про помилки на консоль (небуферизований режим виводу даних)
`clog`	Виведення налагоджувальної або діагностичної інформації на консоль (буферизований режим виводу даних)

У більшості програм на C++ для роботи з консоллю, як правило, використовують лише перші два об'єкти стандартних потоків – `cin` та `cout`. Для роботи з цими об'єктами слід скористатись ***операторами виведення*** `<<` даних у потік або ***операторами введення*** `>>` даних з потоку.

Оператори введення-виведення `>>`, `<<` здатні самостійно розпізнавати тип даних. На відміну від програм мовою C, де для правильного введення/виведення даних необхідно явно вказувати специфікацію типу даних (`%...`), у програмах на C++, що використовують потоки введення-виведення, це непотрібно.

Іншою особливістю операторів `<<`, `>>` є те, що їх можна послідовно використовувати в одному рядку коду, створюючи програмні конструкції як у наступному прикладі:

```
cout << "Hello" << ',' << " world!\n";
```

В цьому прикладі спочатку на консоль буде виведений текстовий рядок Hello, потім символ коми «,», потім ще один текстовий рядок world!, і текстовий курсор перейде на новий рядок (завдяки вказаній escape-послідовності `\n`).

Замість використання escape-послідовності `\n` можна скористатись стандартним ***маніпулятором потоку*** виводу `endl`, який переводить текстовий курсор на наступний рядок. Відповідний код, що використовує `endl`, буде виглядати так:

```
cout << "Hello" << ',' << " world!" << endl;
```

Окрім простого (без аргументів) маніпулятора `endl`, з потоками можуть використовуватись *параметризовані* маніпулятори, які мають принаймні один параметр (вони стають доступні при підключенні файлу заголовку iomanip). Доволі часто в програмах на C++ використовуються такі параметризовані маніпулятори: `setbase(int b)`, `setprecision(int n)`, `setw(int w)`. Усі вони мають один параметр типу `int`. Маніпулятор `setbase(int b)` задає основу системи числення (`b` = 8, 10, 16), яка буде використовуватись потоком при введенні-виведенні числових даних. Маніпулятор `setprecision(int n)` визначає кількість цифр `n` цілої та дробової частини дійсного числа, які будуть враховуватись при введенні числа з консолі або виведенні його на консоль. Останній маніпулятор `setw(int w)` задає ширину поля на консолі в символах, з якого зчитуються дані або в яке записуються дані. Наприклад, код

```
cout << setw(5) << oct << x << endl;
cout << setw(5) << dec << x << endl;
cout << setw(5) << hex << x << endl;
```

виводить число `x` в межах поля шириною 5 символів у вісімковій (маніпулятор `oct` або `setbase(8)`), десятковій (маніпулятор `dec` або `setbase(10)`) та шістнядцятковій (маніпулятор `hex` або `setbase(16)`) системах числення. А код

```
cout << setprecision(5) << d;
```

виводить дійсне число `d` у вигляді наближеного представлення з 5 цифрами. Спочатку виводяться цифри цілої частини числа, потім – дробової частини числа. Наприклад, для `d = 3.1415926`, буде виведено 3.1416, а для `d = 314.15926` – число 314.16.

Наведемо приклад програми, яка виконує ту ж саму послідовність дій, що й програма з лістингу 1.2, проте використовує не функції стандартної бібліотеки C, а потоки введення-виведення C++:

Лістинг 1.4:

```
1: #include <iostream>
2:
```

```
 3: using namespace std;
 4:
 5: int main( void )
 6: {
 7:      int i;
 8:      cout << "Please input integer number: ";
 9:      cin >> i;
10:      cout << "You have entered number " << i;
11:      return 0;
12: }
```

Розглянемо відмінності цієї програми, від програми з лістингу 1.2.

У рядку 1 замість файлу заголовку stdio.h, що входить до складу стандартної бібліотеки С, включається файл заголовку стандартної бібліотеки С++ iostream. Цей файл містить інформація про стандартні потоки введення-виведення С++.

Рядок 3 оголошує, що за замовчуванням використовується стандартний ***простір імен*** std. Цей рядок в програмі є необов'язковим, але його наявність дозволяє трохи спростити текст програми. Це пов'язане з тим, що всі змінні, об'єкти і функції стандартної бібліотеки С++ включені у стандартний простір імен `std` і, завдяки цьому, їх «справжні» імена мають вигляд std::ім'я. Використання конструкції `using namespace std;` дозволяє не писати префікс `std::` перед кожним ідентифікатором, визначеним у стандартній бібліотеці С++. Інакше, якщо видалити рядок 3 у лістингу 1.4, потрібно буде всюди замість `cout`, `cin` писати `std::cout`, `std::cin`.

У рядку 8 використовується оператор виводу `<<` текстового рядку на консоль за допомогою стандартного потоку `cout`. Зчитування даних у змінну `i` цілого типу з `cin` відбувається за допомогою `>>` в рядку 9. Інформація про введене число виводиться через `cout` на консоль у рядку 10.

Використання української мови при роботі з консоллю

Буває так, що спроба вивести на консоль текстове повідомлення, записане кирилицею, за допомогою функції `printf()` або через потік `cout` приводить до появи на екрані незрозумілого набору символів. Це може бути пов'язано з тим, що IDE (наприклад, Microsoft Visual Studio) або бібліотеки С/С++ можуть використовувати для відображення символів української мови одну кодову таблицю (скажімо, cp1251), а в консолі застосовується інша (скажімо, cp866). В результаті, програма передає в консоль коди символів своєї кодової таблиці, які інтерпретуються в термінах іншої кодової таблиці, де тим же самим кодам відповідають зовсім інші символи. Оскільки символи латинського алфавіту в усіх існуючих таблицях символів стандартизовані, при

роботі з англомовними текстами таких помилок, як правило, не виникає. В той же час символи інших алфавітів, навпаки, як правило, розташовуються в різних кодових таблицях у різних місцях (тобто мають різні коди), що й призводить до невірного введення-виведення символів.

Одним із способів вирішення цієї проблеми є перекодування тексту перед виводом його на консоль. Для цього можна застосувати функцію `setlocale()`, яка якраз і виконує перекодування символів. Вона визначена у файлі заголовку locale.h бібліотеки C або clocale бібліотеки C++. Приклад використання цієї функції наведено у лістингу 1.5.

Лістинг 1.5:

```
 1: #include <iostream>
 2: #include <clocale>
 3:
 4: int main( void )
 5: {
 6:     // Виклик функції локалізації.
 7:     setlocale(LC_CTYPE, "ukr");
 8:     std::cout << "Привіт! Ми з України" << endl;
 9:     return 0;
10: }
```

Функція `setlocale()` впливає на всі функції та команди, вказані після її виклику в тексті програми, і має два аргументи. Перший з них – `LC_TYPE` вказує на необхідність перекодувати саме набір символів (можна ще, наприклад, змінити назву валюти чи способи позначення дійсних чисел). Другий аргумент вказує на бажану мову. В нашому випадку це українська мова, якій відповідає текстовий рядок `"ukr"`. Ще можна залишити порожні подвійні лапки `""`, тоді буде використано набір символів, обраний за замовчуванням у встановленій на комп'ютері операційній системі.

Після цього вже можна виводити у консоль повідомлення, написані з використанням символів кирилиці (див. рядок 8 у лістингу 1.5).

Альтернативним способом, доступним для комп'ютерів з ОС Windows, є використання функцій цієї ОС для задання режиму роботи з консоллю. Це можна зробити підключивши файл windows.h за допомогою директиви `#include <windows.h>`. Потім на початку програми достатньо викликати функцію `SetConsoleOutputCP()` для встановлення режиму виводу символів на консоль та/або функцію `SetConsoleCP()` для встановлення режиму введення символів з консолі. Приклад використання цих функцій наведений у лістингу 1.6:

Лістинг 1.6:

```
 1: #include <iostream>
```

```
 2: #include <windows.h>
 3:
 4: int main( void )
 5: {
 6:     SetConsoleOutputCP(1251);
 7:     SetConsoleCP(1251);
 8:
 9:     std::cout << "Привіт! Ми з України" << endl;
10:
11:     return 0;
12: }
```

У рядках 6, 7 лістингу 1.6 функції `SetConsoleOutputCP()` та `SetConsoleCP()` використовуються для встановлення кодової таблиці `1251` (т. зв. кодової таблиці Windows–1251) для введення та виведення символів. За допомогою вказаних функцій можна задати й інші кодові таблиці, наприклад, таблицю символів української мови koi8-u, якій відповідає код `21866`, тощо.

Перевірка умов у програмах на C/C++

При написанні програм часто виникає необхідність перевірки різноманітних умов. Наприклад, якщо програма працює з даними, які користувач вводить з клавіатури, перш ніж починати роботу з цими даними, виявляється доцільним перевірити їх коректність. Якщо користувач ввів дані невірно, бажано принаймні вивести повідомлення про невірно введені дані.

У мові програмування C/C++ для перевірки умов можуть застосовуватись умовні оператори `if`, `if-else`. Їх синтаксис наступний:

```
if( умова ) оператор 1 або блок операторів 1;
else оператор 2 або блок операторів 2;
```

Якщо вказана умова виконується, буде виконуватись тільки оператор 1 або блок операторів 1. Якщо ж умова не виконується, то буде виконуватись оператор 2 або блок операторів 2.

Частина коду з `else` може бути відсутньою. В цьому випадку кажуть, що застосовується умовний оператор `if` (оператор без блоку `else`). Якщо ж присутній блок коду, пов'язаний із `if` та з `else`, то в програмі використовується умовний оператор `if-else`.

Оператори 1 та 2 мають бути *простими* операторами. Якщо необхідно використати декілька простих операторів, їх об'єднують в ***блок операторів***,

заключаючи прості оператори у фігурні дужки {}.

Приклад, використання операторів if, if-else наведено в лістингу 1.7.

Лістинг 1.7:

```
 1: #include <iostream>
 2: #include <clocale>
 3:
 4: using namespace std;
 5:
 6: int main( void )
 7: {
 8:     int age;
 9:
10:     setlocale(LC_CTYPE,"ukr");
11:
12:     cout << "Скільки Вам років? ";
13:     cin >> age;
14:     if( age < 0 )
15:     {
16:         cout << "Невірно введені дані!";
17:         return 0;
18:     }
19:     if( age <= 35 ) cout << "Ви - молодий вчений!";
20:     else cout << "Ви - досвідчений вчений!";
21:     return 0;
22: }
```

Спочатку інформація про вік користувача програми зберігається у змінній age (рядок 13). Далі за допомогою оператора if перевіряється коректність введених даних (рядок 14). Якщо дані некоректні, то виконується код рядків 16, 17 і робота програми завершується. Якщо ж дані введені коректно, то виводиться інформація про користувача, залежно від введеного ним віку (рядки 19-20).

Завдання

Варіант 1

1) Створити змінну типу double з ім'ям R (радіус кола).
2) Створити константу типу double з ім'ям Pi, якій присвоїти значення 3.14159.
3) Використовуючи функцію scanf() або scanf_s() задати значення

змінної R.

4) Реалізувати перевірку значення R та вивести повідомлення про помилку – текстовий рядок Error, якщо R < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо R > 0 програма має виконати пункти 5, 6.
5) Розрахувати периметр та площу кола радіусом R, використавши константу Pi і записавши результат у змінні типу double з іменами P та S.
6) Використовуючи функцію printf(), вивести на екран: формули, за якими розраховувалися величини P та S, та їх значення з точністю 2 знаки після коми.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 2

1) Створити змінні типу int з іменами L (довжина дроту в одиницях [мм]), S (площа поперечного перерізу дроту в одиницях [10^{-2} мм^2]).
2) Задати константу типу double з іменем R0 (питомий опір), якій присвоїти значення 1.68×10^{-8} (в одиницях Ом×м).
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних L та S.
4) Реалізувати перевірку значень L та S, вивести повідомлення про помилку – текстовий рядок Error, якщо L < 0, S < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Інакше, якщо L, S > 0 програма має виконати пункти 5, 6.
5) Розрахувати опір дроту в [Ом] довжиною L [мм] та площею поперечного перерізу S [10^{-2} мм^2], записавши результат у змінній double R.
6) Використовуючи функцію printf(), вивести на екран: формулу для розрахунку опору R, значення з точністю 3 знаки після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 3

1) Створити змінні типу float з іменами D (відстань між обкладками плоского конденсатора), S (площа обкладок конденсатора).
2) Задати константи float Epsilon, Epsilon0 (діелектрична проникність та електрична стала), яким присвоїти значення 10 та 8.85418×10^{-12} (в одиницях Ф/м).
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних D та S.

4) Реалізувати перевірку значень D та S, вивести повідомлення про помилку – текстовий рядок Error, якщо $D^2 > S/10$ (критерій плоского конденсатора, в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо $D^2 < S/10$ програма має виконати пункти 5, 6.
5) Розрахувати ємність плоского конденсатора з площею обкладинок S та відстанню між ними D, записавши результат у змінній C типу double.
6) Використовуючи функцію printf(), вивести на екран: формулу для розрахунку ємності C, значення з точністю 5 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 4

1) Створити змінні типу long int з іменами A, T.
2) Задати константу типу float з ім'ям Pi, якій присвоїти значення 3.14159.
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних A та T, які мають наближено дорівнювати відстані від Землі до Сонця в кілометрах та часу обертання Землі навколо Сонця в секундах.
4) Реалізувати перевірку значень A та T, вивести повідомлення про помилку – текстовий рядок Error, якщо $A < 10^8$ та $A > 2\times10^8$ (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо $10^8 < A < 2\times10^8$ програма має виконати пункти 5, 6.
5) Розрахувати лінійну та кутову швидкість обертання Землі навколо Сонця, якщо радіус орбіти A і період обертання T, записавши результат у змінних float V та W відповідно.
6) Використовуючи функцію printf() вивести на екран: формули для розрахунку швидкостей, їх значення з точністю 4 знаків після коми та розмірності отриманих величин.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 5

1) Створити змінні типу double з іменами T1, T2, L.
2) Задати константу float Lambda (коефіцієнт теплопровідності матеріалу), якій присвоїти значення 0.1 (в одиницях Вт/м·К).
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних T1, T2 та L.
4) Реалізувати перевірку значень T1 та T2, L, вивести повідомлення про помилку – текстовий рядок Error, якщо T1 > T2 або L <= 0 (в цьому випадку

пункти 5, 6 не мають виконуватись програмою). Якщо T2 > T1, L > 0 програма має виконати пункти 5, 6.

5) Розрахувати питомий тепловий потік `Q` між двома поверхнями, що знаходяться на відстані `L` з температурами `T1` та `T2` між якими знаходиться матеріал з коефіцієнтом теплопровідності `Lambda` (формула для розрахунку Q = – Lambda×(T2-T1)/L), записавши результат у змінній `float Q`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку питомого теплового потоку `Q`, значення з точністю 3 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 6

1) Створити змінну типу `double` з іменем `m` (маса тіла).
2) Задати константу типу `double` з іменем `c`, якій присвоїти значення швидкості світла у вакуумі (в одиницях м/с).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінній `m`.
4) Реалізувати перевірку значень `m`, вивести повідомлення про помилку – текстовий рядок Error, якщо m < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо m > 0 програма має виконати пункти 5, 6.
5) Використовуючи формулу Альберта Ейнштейна $E = m \cdot c^2$, де m – маса, c – швидкість світла, визначити енергію E об'єкту масою `m`, записавши результат у змінну `double E`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку енергії `E`, значення з точністю 3 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 7

1) Створити змінну типу `float` з іменем `Lambda`.
2) Задати константи типу `double` з іменами `h`, `c`, яким присвоїти значення 6.62607×10^{-34} (Дж×с, стала Планка) та 2.99792×10^{8} (м/с, швидкість світла у вакуумі).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `Lambda`.
4) Реалізувати перевірку значень `Lambda`, вивести повідомлення про помилку – текстовий рядок Error, якщо Lambda < 0 (в цьому випадку пункти 5,

6 не мають виконуватись програмою). Якщо Lambda > 0 програма має виконати пункти 5, 6.

5) Виходячи з формули для енергії кванта (E = h×f) розрахувати частоту f та енергію кванта E електромагнітного випромінювання з довжиною хвилі Lambda. Результат записати у змінні double f та E, відповідно.
6) Використовуючи функцію printf() вивести на екран: формули для розрахунку частоти f та енергії E, значення з точністю 3 знаків після коми та розмірності отриманих величин.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 8

1) Створити змінні типу float з іменами Q, R.
2) Задати константу типу float з іменем K, якій присвоїти значення 8.98774×10^9 ($Н\times м^2/Кл^2$, стала Кулона).
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних Q, R.
4) Реалізувати перевірку значень R, вивести повідомлення про помилку – текстовий рядок Error, якщо R ≤ 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо R > 0 програма має виконати пункти 5, 6.
5) Вважаючи, що два точкових тіла мають однакові заряди Q, які знаходяться на відстані R визначити силу з якою відштовхуються заряди, записавши результат у змінну double F.
6) Використовуючи функцію printf() вивести на екран: формулу для розрахунку сили відштовхування F, значення з точністю 2 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 9

1) Створити змінну типу unsigned int з іменем T та змінні типу float з іменами P, V.
2) Задати константу типу double з іменем R, якій присвоїти значення 8.31 (Дж/моль К, універсальна газова стала).
3) Використовуючи функцію scanf() або scanf_s() задати значення змінних T, P, V.
4) Реалізувати перевірку значень P, вивести повідомлення про помилку – текстовий рядок Error, якщо P, V < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо P, V ≥ 0 програма має виконати пункти

5, 6.

5) Розрахувати кількість молей газу за формулою $m = \frac{PV}{RT}$, що випливає з рівняння стану ідеального газу та записати результат у змінну `double m`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку кількості молей газу, значення `m` з точністю 5 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 10

1) Створити змінну типу `unsigned int` з іменем `T` та змінні типу `float` з іменами `V`, `m`.
2) Задати константу типу `double` з іменем `R`, якій присвоїти значення 8.31 (Дж/моль К, універсальна газова стала).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінних `T`, `m`, `V`.
4) Реалізувати перевірку значень `V`, вивести повідомлення про помилку – текстовий рядок `Error`, якщо `V < 0` (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо `V ≥ 0` програма має виконати пункти 5, 6.
5) Розрахувати тиск газу за формулою $P = \frac{mRT}{V}$, що випливає з рівняння стану ідеального газу та записати результат у змінну типу `double` з іменем `P`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку тиску газу, значення `P` з точністю 4 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 11

1) Створити змінну типу `unsigned int` з іменем `T` та змінні типу `float` з іменами `P`, `m`.
2) Задати константу типу `double` з іменем `R`, якій присвоїти значення 8.31 (Дж/моль К, універсальна газова стала).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінних `T`, `m`, `P`.
4) Реалізувати перевірку значень `m`, вивести повідомлення про помилку –

текстовий рядок Error, якщо m < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо m ≥ 0 програма має виконати пункти 5, 6.

5) Розрахувати об'єм ідеального газу за формулою $V = \frac{mRT}{P}$, що випливає з рівняння стану ідеального газу та записати результат у змінну `float V`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку об'єму газу, значення `V` з точністю 5 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 12

1) Створити змінні типу `float` з іменами `P`, `V`, `m`.
2) Задати константу типу `double` з іменем R, якій присвоїти значення 8.31 (Дж/моль К, універсальна газова стала).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінних `P`, `V`, `m`.
4) Реалізувати перевірку значень `P`, вивести повідомлення про помилку – текстовий рядок Error, якщо P < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо P ≥ 0 програма має виконати пункти 5, 6.
5) Розрахувати температуру ідеального газу за формулою $T = \frac{PV}{Rm}$, що випливає з рівняння стану ідеального газу та записати результат у змінну `double T`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку кількості молей газу, значення `T` з точністю 6 знаків після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 13

1) Створити змінну типу `int` з іменем `X`.
2) Задати константи типу `int` з іменами `A`, `B`, `C`, `D` яким присвоїти значення 3, 2, 4, 1.
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `X`.
4) Реалізувати перевірку значень `X`, вивести повідомлення про помилку –

текстовий рядок Error, якщо X < 0 або X >10 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо 0 <= X <= 10 програма має виконати пункти 5, 6.

5) Використавши формулу для стандартного вигляду поліному n-ї степені $Y(X) = A \cdot X^n + B \cdot X^{n-1} + C \cdot X^{n-2} + \ldots$ визначити попередньо n, спираючись на кількість заданих в завданні констант, та знайти значення *Y*, записавши результат у змінну `long int Y`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для визначення Y(X) та значення величини `Y`.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 14

1) Створити змінну типу `int` з іменем `T`.
2) Задати константу типу `double G`, якій присвоїти значення 9.8 (м/с^2, прискорення вільного падіння).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `T`.
4) Реалізувати перевірку значень `T`, вивести повідомлення про помилку – текстовий рядок Error, якщо T < 0 (в цьому випадку пункти 5, 6 не мають виконуватись програмою). Якщо T ≥ 0 програма має виконати пункти 5, 6.
5) Розрахувати відстань, яку пролетить тіло в гравітаційному полі біля поверхні Землі за заданий час `T`, вважаючи, що його початкова швидкість рівна нулеві, записавши результат у змінну `double H`.
6) Використовуючи функцію `printf()` вивести на екран: формулу для розрахунку відстані `H`, значення з точністю 1 знак після коми та розмірність отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 15

1) Створити змінну типу `long int` з іменем `X`.
2) Задати константи `double A, B, C`, яким присвоїти значення 2, –5 та –8.
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `X`.
4) Реалізувавши перевірку значень, вивести повідомлення про помилку – текстовий рядок Error, якщо значення $AX^2 + BX + C$ менше нуля.
5) Обчислити та вивести за допомогою `printf()` на екран значення виразу $Y(X) = \sqrt{AX^2 + BX + C}$ для введеного числа `X`, записавши результат у

змінну `float Y`.

6) Створити версію програми, де пункти 3, 4 та 5 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 16

1) Створити змінну типу `float` з іменем `m`, та змінну типу `double` з іменем `E`.
2) Задати константи `double c`, `h`, яким присвоїти значення 299792458 (м/с, швидкість світла у вакуумі) та 6.62607×10^{-34} (Дж×с, стала Планка).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `m` (у кг).
4) Реалізувати перевірку значень `m`, вивести повідомлення про помилку – текстовий рядок `Error`, якщо значення введеного числа від'ємне.
5) Обчислити значення комптонівської довжини хвилі тіла за формулою $\Lambda = h / mc$ і зберегти його у змінній `double Lambda`.
6) Отриманий результат із зазначеною розмірністю вивести на консоль за допомогою функції `printf()` з точністю до 3 знаків після коми.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 17

1) Створити змінну типу `float` з іменем `a` та змінну типу `double` з іменем `m`.
2) Задати константу `int Ro`, якій присвоїти значення 1000 (кг/м^3, густина матеріалу).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `a` (у м).
4) Реалізувати перевірку значень `a`, вивести повідомлення про помилку – текстовий рядок `Error`, якщо значення введеного числа від'ємне.
5) Обчислити значення маси куба з ребром `a`, густина матеріалу куба `Ro`. Результат записати у змінну `m`.
6) Результат вивести використовуючи функцію `printf()` з точністю до 2 знаків після коми із зазначенням розмірності отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 18

1) Створити змінні типу `float` з іменами `m`, `R`, та змінну типу `double` з іменем `V`.

2) Задати константу `double G`, якій присвоїти значення $6.67 \cdot 10^{-11}$ ($м^3/кг\ с^2$, гравітаційна стала).
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінних `m` (у кг) та `R` (у м).
4) Реалізувати перевірку значень `m` та `R`, вивести повідомлення про помилку – текстовий рядок Error, якщо значення одного з цих чисел менше або дорівнює нулю.
5) Обчислити значення першої космічної швидкості $V = \sqrt{G\frac{m}{R}}$ для сферичного тіла масою `m` з радіусом `R` (вважати що радіус орбіти співпадає з радіусом тіла). Результат обчислень зберегти у змінній `double V`.
6) Результат вивести на консоль використовуючи функцію `printf()` з точністю до 5 знаків після коми із зазначенням розмірності отриманої величини.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 19

1) Створити змінні типу `float` з іменами `S`, `R`, та змінну типу `int` з іменем `n`.
2) Задати константу `double Pi`, якій присвоїти значення 3.14.
3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінних `S` (у м) та `R` (у м).
4) Реалізувати перевірку значень `R`, `S`, вивести повідомлення про помилку – текстовий рядок Error, якщо значення будь-якого із введених чисел менше або дорівнює нулю.
5) Обчислити повну кількість обертів `n` колеса радіусом `R`, яке пройшло відстань `S`. Ціла кількість обертів становить $n = \left[\frac{S}{2\pi R}\right]$.
6) Використовуючи функцію `printf()` вивести формулу для розрахунку `n` та значення кількості обертів `n`.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

Варіант 20

1) Створити змінну типу `int` з іменем `V` , та змінну типу `float` з іменем `n`.
2) Задати константу `double c`, якій присвоїти значення 299792458 (м/с, швидкість світла у вакуумі).

3) Використовуючи функцію `scanf()` або `scanf_s()` задати значення змінної `n`.
4) Реалізувати перевірку значень `n`. Вивести повідомлення про помилку – текстовий рядок Error, якщо значення введеного числа менше одиниці.
5) Обчислити швидкість світла у матеріалі з показником заломлення `n`, результат записати в змінну `float V`. Формула для розрахунку $V = \frac{c}{n}$.
6) Використовуючи функцію `printf()` вивести формулу для розрахунку та значення швидкості (у м/с) з точністю 3 знаки після коми.
7) Створити версію програми, де пункти 3, 4 та 6 були б реалізовані за допомогою потоків введення-виведення C++.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Яка структура програми мовою C/C++?
2. Для чого використовуються коментарі? Які бувають форми запису коментарів у програмах мовами C/C++?
3. Яким чином описуються змінні в програмі на C/C++?
4. Чим змінні відрізняються від констант? Наведіть приклад визначення та використання константи в коді програми.
5. Які базові типи даних (змінних) доступні у мові програмування C/C++?
6. Що таке керуючі символи (escape-послідовності)? Наведіть приклади їх використання.
7. Які типи операцій існують в C/C++? Які операнди при цьому використовуються?
8. Які арифметичні операцій використовуються для числових типів у мові C/C++? Наведіть приклади.
9. Яким чином в C здійснюється введення даних з клавіатури та виведення даних на консоль?
10. Яким чином в C++ здійснюється введення даних з клавіатури та виведення даних на консоль?
11. Для чого потрібні маніпулятори потоків C++? Наведіть приклади їх використання.
12. Що таке умовний оператор `if`, `if-else`? Наведіть приклади його використання.

Лабораторна робота № 2.
Базові програмні конструкції мови C/C++

Мета роботи:

1) Оволодіти практичними навичками використання умовних операторів `if-else` та оператора вибору `switch`.
2) Навчитися застосовувати циклічні оператори `for`, `while`, `do-while`.

ЛОГІЧНІ ВИРАЗИ ТА ЛОГІЧНІ ОПЕРАТОРИ МОВИ C/C++

Логічним (булевим) виразом або ***висловлюванням*** називається твердження, якому завжди можна поставити у відповідність одне і тільки одне з двох логічних значень: ***хибність*** або ***істина***. В комп'ютерній математиці істина часто позначається як «1» (логічна одиниця), а хибність – як «0» (логічний нуль).

Мова програмування С має механізми для роботи з логічними виразами (наприклад використання операторів `if`, `if-else` вже частково розглядалось в лабораторній роботі 1). Проте у мові С **не передбачено** окремого булевого типу даних, а також булевих констант для значень *хибність* та *істина*. Отже, у С робота з логічними виразами та логічними змінними (які зберігають результат обчислень логічного виразу) має бути реалізована за допомогою інших (небулевих) типів даних. Для цього може використовуватись довільний тип цілих чисел, проте, традиційно використовують тип `int`.

У мові С значення будь-якої змінної або виразу вважається істинним, якщо воно відрізняється від 0, і хибним, якщо воно строго дорівнює 0. Таким чином, логічна одиниця «1» (істина) у мові С може бути інтерпретована як *будь-яка ненульова величина*, а хибність «0» – як *нульова величина*. Оператори, які обчислюють логічні вирази (див. нижче), як правило, повертають число `1`, якщо результатом виразу є *істина*, і `0`, якщо *хибність* (для таких операторів «1» має значення `1`, а «0» має значення `0`).

Натомість у C++ визначено спеціальний логічний тип даних `bool`. Змінна типу `bool` займає у пам'яті 1 байт і може приймати лише два можливих значення: `true` (істина, логічна одиниця «1») або `false` (хибність, логічний нуль «0»).

У мові C/C++ визначені наступні логічні операції:

Унарні операції (тобто операції, які виконуються над одним операндом):

Заперечення (логічне "НЕ") – унарна операція, результатом якої є логічне значення, протилежне до значення операнду (логічної змінної або виразу),

до якого вона була застосована. У мові C/C++ позначається символом ! перед операндом. Наприклад, !x означає заперечення x.

Бінарні операції (для двох операндів):

Кон'юнкція (логічне "І") – бінарна операція, результатом якої є *істина*, коли обидва операнди (логічні змінні або вирази) є істинними одночасно і *хибність* в усіх інших випадках. У мові C/C++ логічне "І" позначається символами &&, наприклад, x && y означає x "І" y.

Диз'юнкція (логічне "АБО") – бінарна операція, результатом якої є *хибність* тоді і тільки тоді, коли хибні одночасно обидва операнди (логічні змінні або вирази). В усіх інших випадках результатом виконання операції є істина. У мові C/C++ логічне "АБО" позначається символами ||, наприклад, x || y означає x "АБО" y.

Операндами наступних бінарних операцій є змінні, вирази або результати роботи функцій, що відносяться до одного із стандартних типів даних C або C++:

Дорівнює – бінарна операція, результатом якої є *істина* тоді і тільки тоді, коли значення обох операндів є однаковими. У мові C/C++ операція «дорівнює» позначається двома символами рівності == (не сплутайте з оператором присвоювання =, який записується за допомогою єдиного символу =) . Наприклад, код x == y означає x дорівнює y. Його результатом є «1», якщо величини x та y рівні одна одній, інакше результатом є «0».

Не дорівнює – бінарна операція, результатом якої є *істина*, тоді і тільки тоді, коли значення обох операндів є різними. У тексті програми на C/C++ операція «не дорівнює» позначається символами !=. Наприклад, код x != y означає x не дорівнює y. Його результатом є «1», якщо величини x та y не рівні між собою, інакше результатом є «0».

Менше – бінарна операція, результатом якої є *істина*, тоді і тільки тоді, коли значення операнда, що стоїть зліва є меншим за значення операнда, що стоїть справа. ***Більше*** – бінарна операція, результатом якої є *істина*, тоді і тільки тоді, коли значення операнда, що стоїть зліва є більшим за значення операнда, що стоїть справа. В програмах на C/C++ ці оператори позначаються символами < та >. Наприклад, вираз x > y є істинним, якщо величина x є більшою за y. При цьому вираз x < y буде хибним.

Менше або дорівнює – бінарна операція, результатом якої є *істина*, тоді і тільки тоді, коли значення операнда, що стоїть зліва є не більшим за значення операнда, що стоїть справа. ***Більше або дорівнює*** – бінарна операція, результатом якої є *істина*, тоді і тільки тоді, коли значення операнда, що стоїть зліва є не меншим за значення операнда, що стоїть справа. Зазначені оператори позначаються комбінацією з двох символів <= або >=. Наприклад, вираз x >= y буде істинним, якщо величина x є більшою за y або дорівнює y.

Тернарні операції (з трьома операндами):

Умовний оператор `F ? a : b` результатом якого буде значення виразу `a`, якщо умова `F` істинна і `b` – якщо умова `F` хибна. Даний оператор є скороченою формою запису оператора `if-else`.

Табл. 2.1. Логічні операції у мові C/C++ (змінна `a` буде містити результат обчислення логічного виразу – «1» або «0»)

Позначення	Зміст	Приклад коду
`!`	Логічне заперечення	`a = !x;`
`&&`	Логічне "І"	`a = x && y;`
`||`	Логічне "АБО"	`a = x || y;`
`==`	Дорівнює	`a = x == y;`
`!=`	Не дорівнює	`a = x != y;`
`<`	Менше	`a = x < y;`
`>`	Більше	`a = x > y;`
`<=`	Менше або дорівнює	`a = x <= y;`
`>=`	Більше або дорівнює	`a = x >= y;`
`?:`	Умовний вибір з двох виразів	`a = y > x ? y : x;`

УМОВНІ ОПЕРАТОРИ ТА ОПЕРАТОР ВИБОРУ

Структура керування виконанням програми, що виконує один із заздалегідь описаних алгоритмів (команду або перелік команд), залежно від виконання або невиконання деякої умови, називається ***розгалуженням***. У мові C/C++ розгалуження можуть реалізовуватись за допомогою умовних операторів `if`, `if-else`, оператора вибору `switch` або оператора `?:`, розглянутого раніше.

Умовні оператори if-else та if

Одним із способів реалізації розгалужень у мові C/C++ є ***умовний оператор***, що має такий синтаксис

```
if( умова ){ Оператор 1; } else { Оператор 2; }
```

При цьому Оператор 1 буде виконуватися у випадку істинності умови, що стоїть в дужках після оператора `if`, а Оператор 2 – у випадку хибності цієї умови. У випадку, якщо Оператор 1 або Оператор 2 складається тільки з однієї команди (простий оператор), фігурні дужки `{}` навколо відповідного

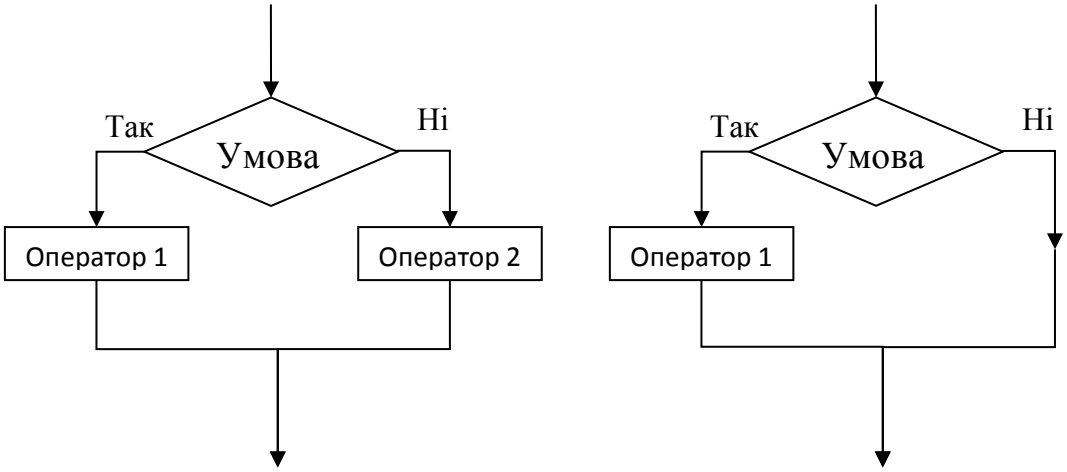
оператора можна не писати.

Таким чином, умовний оператор `if-else` реалізує розгалуження між двома алгоритмами, які формально представлені Операторами 1 та 2.

В якості умови може виступати не лише логічний вираз, а в принципі, будь-яка змінна, вираз або значення функції. Якщо відповідна величина приймає ненульове значення, це буде інтерпретуватися як істинність умови. Якщо нульове значення – як її хибність.

Зазначимо, що блок розгалуження з `else` є необов'язковим. Його можна пропускати, якщо у випадку хибності умови програма не повинна виконувати жодних дій.

Блок-схеми роботи оператора `if-else` та оператора `if` (без блоку `else`) показано нижче:



У лістингу 2.1 наведено текст програми, яка за рахунок використання оператора `if-else` (рядки 8–9) буде виводити на консоль лише найбільше з двох дійсних чисел `x` та `y`.

Лістинг 2.1:

```
 1: #include <stdio.h>
 2:
 3: int main( void )
 4: {
 5:     double x = 3.0;
 6:     double y = 1.7;
 7:
 8:     if( x < y ) printf("x < y and y = %f\n", y);
 9:     else printf("x >= y and x = %f\n", x);
10:
11:     return 0;
12: }
```

Оператор вибору switch

Оператор вибору `switch` реалізує розгалуження між довільною кількістю операторів («алгоритмів», блоків коду), кожному з яких відповідає свій блок `case`. Синтаксис цього оператора наступний:

Лістинг 2.2:

```
 1: switch( Умова )
 2: {
 3:     case Константа 1:
 4:     Оператор 1;
 5:     break;
 6:
 7:     case Константа 2:
 8:     Оператор 2;
 9:     break;
10:
11:     // Інші блоки case...
12:
13:     default:
14:     Оператор N;
15:     break;
16: }
```

Умова має бути виразом цілочисельного типу (скажімо, `int`) або типу, що однозначно конвертується в ціле число (наприклад, типу `char`).

Оператор `switch` працює наступним чином. Якщо зазначена Умова відповідає цілочисельному типу, здійснюється порівняння цієї Умови зі значенням кожної із Констант, вказаних після `case`. Якщо знаходиться співпадіння між Умовою та однією із Констант, то виконується команда (чи набір команд) з відповідного блоку. Наприклад, якщо Умова дорівнює Константі 2, при виконанні оператора `switch` буде виконано лише Оператор 2.

Оператор `break` в кінці кожного блоку `case` є необхідним. Якщо його не буде, програма продовжить виконувати всі оператори, розташовані нижче, до блоку `default` включно.

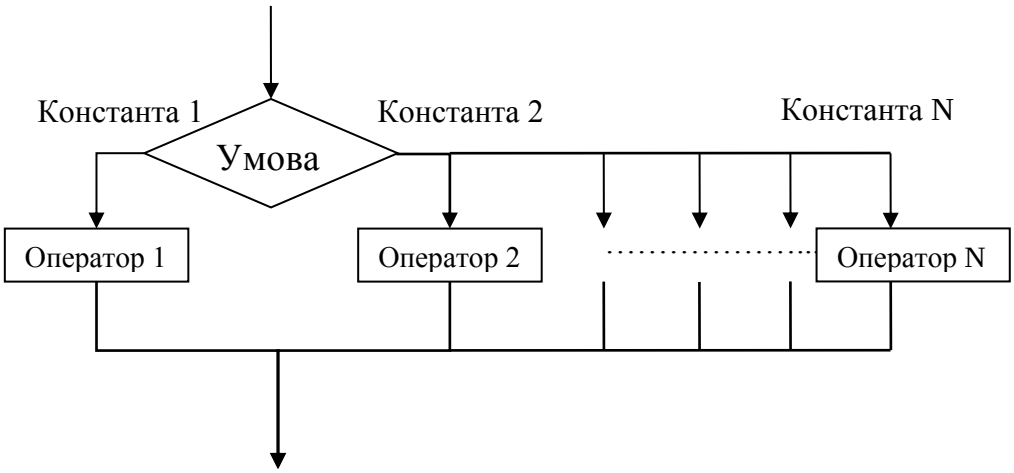
Якщо ж співпадіння між Умовою та однією із наведених Констант немає, виконується набір команд, у блоці, що позначений як `default`. Разом з тим, блок `default` не є обов'язковим. Якщо він відсутній, а співпадіння з жодною константою не буде знайдено, оператор `switch` просто завершить роботу не виконавши жодного з Операторів 1, 2, … N.

Як приклад використання оператора `switch` наведемо фрагмент програми, яка виводить на консоль назву місяця року за номером місяця, введеним з клавіатури.

Лістинг 2.3:

```
 1:  int num;
 2:
 3:  cout << "Будь ласка введіть номер місяця (1-12): ";
 4:  cin >> num;
 5:
 6:  switch( num )
 7:  {
 8:       case 1:
 9:       cout << "Січень" << endl;
10:       break;
11:
12:       case 2:
13:       cout << "Лютий" << endl;
14:       break;
15:
16:       // Інші блоки case для 3-12...
17:
18:       default:
19:       cout << "Помилка" << endl;
20:  }
```

Типова блок-схема роботи оператора `switch` показана нижче.



Оператори циклів

Структура керування виконанням програми, що може здійснювати багаторазове повторення певної послідовності команд, називається ***циклом***. Цикли умовно поділяють на два типи: ***цикли за умовою*** і ***цикли з лічильником***.

Як правило, цикли з лічильником використовують для повторення певної послідовності команд певну, наперед задану кількість разів. На відміну від циклу з лічильником, цикл за умовою виконується не задану кількість разів, а до тих пір, поки задана умова є істинною.

Цикли за умовою while та do-while

У мові C/C++ є два типи циклів за умовою: ***цикл з передумовою*** `while` з синтаксисом

```
while( Вираз ) { Оператор; }
```

та ***цикл з післяумовою*** `do-while` із синтаксисом

```
do { Оператор; } while( Вираз );
```

Оператор, що записаний в фігурних дужках `{}` називають ***тілом циклу***.

Обидва варіанти циклів виконують набір інструкцій в тілі циклу, доки Вираз в дужках залишається істинним. Особливістю синтаксису циклу `do-while` є те, що він має завершуватись крапкою з комою `;`.

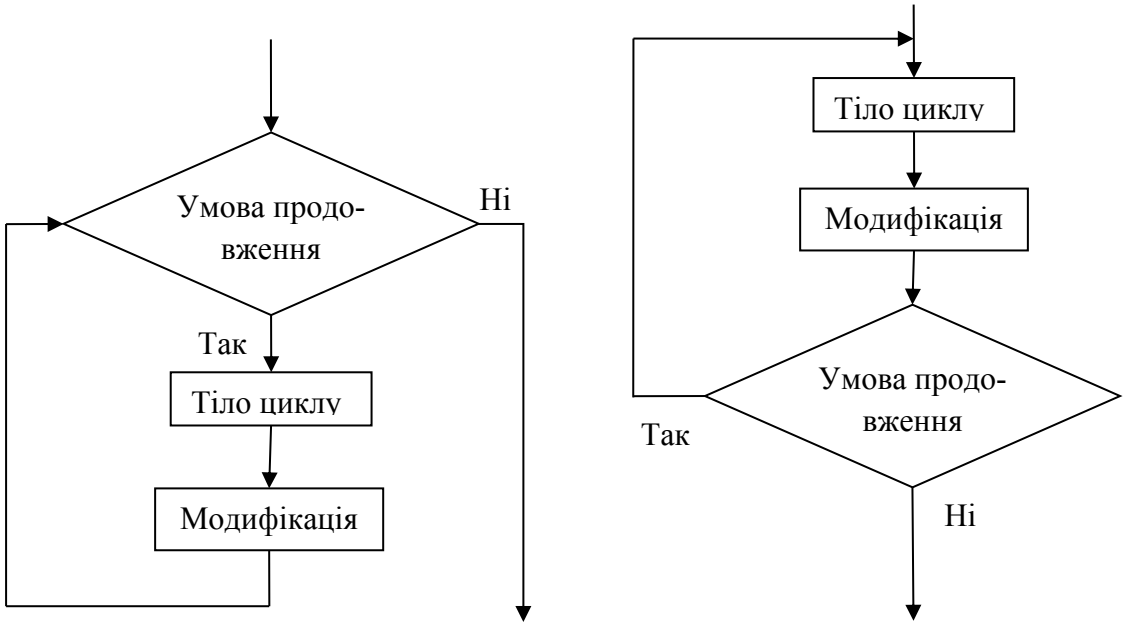
Різниця між циклами `while` та `do-while` полягає в тому, що для циклу з передумовою `while` перевірка виразу здійснюється до першого виконання тіла циклу. Для циклу з післяумовою `do-while` перевірка умови здійснюється вже після того, як виконається перший раз тіло циклу. Таким чином, тіло циклу `while` може взагалі бути не виконаним, в той час як тіло циклу `do-while` гарантовано буде виконано мінімум 1 раз.

Будь-який алгоритм можна реалізувати за допомогою як циклу `while`, так і циклу `do-while`. На практиці частіше зустрічаються програми з циклом `while`. Цикли з післяумовою здебільшого використовують у випадках, коли умова є невизначеною до початку виконання тіла циклу.

В якості Виразу – умови повторення циклу – може бути використаний довільний логічний вираз, а також будь-яка функція чи величина, результат якої є нульовим або ненульовим числом.

Важливо відзначити, що в тілі циклу має бути передбачена модифікація виразу-умови, щоб цикл міг завершитися на одному із кроків. Якщо цього не зробити, цикл стане нескінченим, що призведе до «зациклювання» програми.

Блок-схема роботи операторів `while` та `do-while` показана нижче.



Цикл з лічильником for

Суттєвою відмінністю циклів із лічильником від циклів за умовою є можливість використання окремої змінної, яка називається ***лічильником***. Цикл з лічильником типово використовується для виконання блоку коду з наперед відомою або заданої кількістю повторень (ітерацій). Такий спосіб виконання коду часто називають ***ітераційним***, оскільки він ґрунтується на виконанні коду протягом певної кількості ітерацій.

У мові C/C++ цикл з лічильником називається циклом `for`[1]. Синтаксис цього циклу наступний:

```
for( Вираз1; Вираз2; Вираз3 ){ Тіло циклу; }
```

Вирази 1, Вираз 2, Вираз 3, що стоять у дужках, типово включають в себе змінну циклу (змінну-лічильник), значення якої змінюється в процесі виконання циклу. За значенням цієї змінної або залежної від неї величини визначається, чи буде цикл повторюватись ще раз, чи його виконання буде припинено.

Вираз 1 є оператором, який має виконуватись перед початком циклу. Зазвичай, у Виразі 1 відбувається ініціалізація змінної-лічильника – присвоєння

[1] Синтаксис циклу `for` полегшує введення та використання змінної-лічильника. Однак введення цієї змінної при записі циклу `for` не є принципово обов'язковим. Формально, можна написати цикл `for`, де змінна-лічильник не буде використовуватись взагалі. Також зазначимо, що цикли `while` та `do-while` можна записати так, щоб в них теж фігурувала змінна-лічильник. Отже, всі три цикли – `while`, `do-while` та `for` – можуть використовувати змінні-лічильники, так і працювати без них. Проте синтаксис мови C/C++ лише для циклу `for` передбачає зручні засоби для роботи із змінними-лічильниками.

їй певного початкового значення.

Вираз 2 визначає умову продовження циклу. Якщо вона виконується, цикл продовжується, інакше – циклу завершує свою роботу.

Вираз 3 є оператором, який має виконуватись в кінці кожної ітерації циклу. Як правило, Вираз 3 реалізує операцію зміни лічильника після кожного проходження циклу.

На кожній ітерації циклу має виконуватись Тіло циклу, яке являє собою один або декілька операторів. Тіло циклу типово записують в фігурних дужках `{}`. Проте, якщо це тіло являє собою простий оператор, фігурні дужки можна не писати.

Алгоритм виконання циклу з лічильником `for` наступний:

1) згідно Виразу 1 виконується початкова ініціалізація (зазвичай, ініціалізація змінної-лічильника);

2) перевіряється істинність умови виконання циклу – Виразу 2;

3) якщо умова хибна, то цикл завершується і відбувається перехід на наступний оператор після циклу;

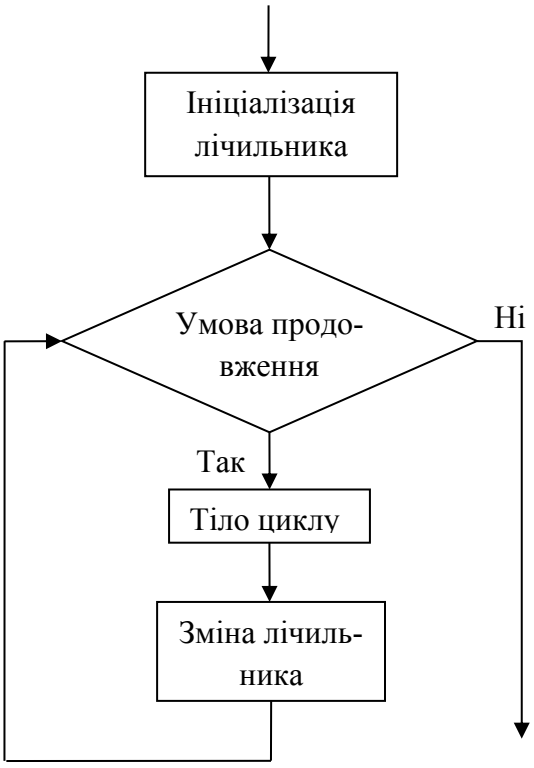
4) якщо умова істинна, то:

– виконується Тіло циклу;

– виконується Вираз 3 (як правило, це операція зміни лічильника);

– алгоритм повторюється ще раз, починаючи з пункту 2).

Ось блок-схема, яка ілюструє типовий алгоритм роботи циклу `for`:



Для прикладу, наведемо фрагмент програми, яка виводить на екран усі натуральні числа від 1 до 100:

Лістинг 2.4:

```
1:  for( int i = 1; i <= 100; i++ )
2:  {
3:      printf("%d \n", i);
4:  }
```

У даній програмі використовується змінна-лічильник `int i`, яка перед початком циклу ініціалізується значенням 1 (перший оператор в дужках після `for` у рядку 1). Тіло циклу (рядок 3) має виконуватись, поки значення `i` менше або дорівнює 100 на що вказує умова `i <= 100` в рядку 1. Під час кожної ітерації циклу викликається функція `printf()`, яка виводить на консоль поточне значення лічильника `i`. В кінці кожної ітерації циклу значення лічильника `i` збільшується на одиницю завдяки операції інкременту `i++`.

Оскільки тіло розглянутого циклу складається всього з одного простого оператора – оператора виклику функції `printf()`, наведену вище програму за бажанням можна записати у вигляді єдиного рядка коду без використання дужок `{}`:

```
for( int i = 1; i <= 100; i++ ) printf("%d \n", i);
```

Оператори переривання виконання коду

Досить часто трапляються ситуації, коли виконання певного блоку коду необхідно перервати, незалежно від будь-яких умов. Для цього у мові C/C++ використовуються оператори `break`, `continue`, `return` та `goto`.

Оператор break

Оператор `break` перериває виконання оператора `while`, `do-while`, `for` або `switch`, в якому він безпосередньо знаходиться. Приклад використання цього оператора наведено в наступному фрагменті програми, який є дещо модифікованою версією програми з лістингу 2.4:

Лістинг 2.5:

```
1:  for( int i = 1; i <= 100; i++ )
2:  {
3:      printf("%d \n", i);
4:      if( i == 77 ) break;
5:  }
```

Код цієї програми містить додатковий оператор (рядок 4) у тілі циклу, який перевіряє чи дорівнює значення лічильника `i` числу 77. Якщо це так, то виконання циклу `for` переривається за допомогою оператора `break`.

Завдяки рядку 4 у лістингу 2.5, програма буде почергово виводити на консоль натуральні числа від 1 до 77. Після цього виконання циклу буде перервано, хоча в самій умові циклу `i <= 100` передбачено, що цикл мав би виконуватись і для значень змінної `i` від 78 до 100.

Оператор continue

Оператор `continue` може застосовуватись тільки в циклах `while`, `do-while` або `for`. Він зупиняє виконання тіла циклу, в якому він безпосередньо знаходиться, і змушує розпочати виконання тіла циклу знову. По суті, цей оператор реалізує переривання поточної ітерації циклу з одночасним примусовим переходом до наступної ітерації циклу. Лістинг 2.6 демонструє приклад використання оператора `continue`.

Лістинг 2.6:

```
1:  for( int i = 1; i <= 100; i++ )
2:  {
3:       if( i % 10 ) continue;
4:       printf("%d \n", i);
5:  }
```

Завдяки рядку 3 з оператором `continue`, результат роботи циклу у лістингу 2.6 суттєво зміниться порівняно з програмою у лістингу 2.4. Так, якщо оператор `continue` у рядку 3 тіла циклу буде виконуватись, це буде спричиняти перехід на наступну ітерацію циклу. Отже, рядок 4, де викликається функція `printf()`, при цьому буде пропускатись. Умовою виконання оператора `continue` у рядку 3 є те, що остача від ділення `i` на `10` відмінна від нуля – тобто для всіх значень `i` від 1 до 100, не кратних 10, оператор `printf("%d \n", i);` (рядок 4) виконуватись не буде. В результаті на консоль замість послідовності цифр 1, 2, 3, … 100 будуть виведені цифри 10, 20, 30, … 100.

Оператор return

Оператор `return` завершує виконання тіла функції. Цей оператор може використовуватись з необов'язковим параметром, який інтерпретується як значення, яке повертає функція. Приклад використання оператора `return` наведено у лістингу 1. Рядок 8 містить оператор `return 0;`, який змушує

завершити виконання функції `main()` з кодом повернення 0.

Оператор goto

Оператор `goto` здійснює безумовний перехід з однієї ділянки коду на іншу, передаючи керування іншому оператору, поміченому міткою. ***Мітка*** – це ідентифікатор з подальшою двокрапкою `:`, який розміщується перед оператором, що помічається. Приклад застосування оператора `goto` показано нижче:

Лістинг 2.7:

```
1:  Stop:              // Назва мітки
2:  Оператори 1;
3:  goto Stop;         // Оператор переходу
4:
5:  Оператори 2;
```

Дійшовши до оператора `goto` у рядку 3, програма завдяки `goto Stop;` перейде на вказану мітку (в даному випадку – `Stop`), після чого будуть знову виконуватись Оператори 1. При такій формі запису програми Оператори 2 ніколи виконуватись не будуть – відповідний код буде недосяжним (unreachable code).

Оператор `goto` і відповідна мітка повинні знаходитися в межах тіла однієї функції. Перехід за допомогою `goto` між різними функціями заборонений. При цьому перехід може відбуватися як «вперед», так і «назад» по коду програми. Єдине обмеження: при переході «вперед», між оператором `goto` і міткою не повинна відбуватись ініціалізація якої-небудь змінної.

По можливості рекомендується уникати використання оператора `goto` в програмах на C/C++. Основна проблема полягає в тому, що він заплутує порядок виконання коду програми, що ускладнює сприйняття логіки виконання такого коду програмістом. Однак, у деяких частинних випадках, використання оператора `goto` може бути корисним. Наприклад, іноді оператор `goto` може використовуватися як альтернатива оператору `break`. Оператор `break` здійснює вихід тільки з одного рівня циклів або оператора `switch`, в той час як за допомогою `goto` можна одразу ж вийти із багаторівневого (вкладеного) циклу або коду із `switch` в потрібне місце програми.

Завдання

Варіант 1

1) Використовуючи один із циклічних операторів, знайти суму натуральних чисел від 1 до N, тобто $1+2+3+...+N$ (число N задається введенням з клавіатури). Записати результат у змінну A.
2) Порівняти отримане число з теоретично виведеною формулою $\frac{N(N+1)}{2}$. Записати результат у змінну B.
3) Використовуючи умовний оператор, порівняти отримані числа. Якщо A = B вивести на екран фразу «Дорівнюють», якщо A > B – «Сума більша», A < B – «Сума менша».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор goto. Приклад:

```
Мітка:
Тіло циклу;
if( умова ) goto Мітка;
```

Варіант 2

1) Використовуючи один із циклічних операторів, обчислити суму квадратів натуральних чисел від 1 до N, тобто $1^2+2^2+3^2+...+N^2$ (число N задається введенням з клавіатури). Записати результат у змінну A.
2) Порівняти отримане число з теоретично виведеною формулою $\frac{N(N+1)(2N+1)}{6}$. Записати результат у змінну B.
3) Використовуючи умовний оператор, порівняти отримані числа. Якщо A = B вивести на екран фразу «Дорівнюють», якщо A > B – «Сума більша», A < B – «Сума менша».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор goto. Див. приклад з варіанту 1.

Варіант 3

1) Використовуючи один із циклічних операторів, знайти безпосереднім розрахунком суму ряду, тобто $2^0+2^1+2^2+...+2^{N-1}$ (число N задається введенням з клавіатури). Записати результат у змінну A.
2) Порівняти отримане число з теоретично виведеною формулою 2^N-1. Записати результат у змінну B.

3) Використовуючи умовний оператор, порівняти отримані числа. Якщо A = B вивести на екран фразу «Дорівнюють», якщо A > B – «Сума більша», A < B – «Сума менша».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 4

1) Використовуючи один із циклічних операторів, знайти безпосереднім розрахунком значення факторіалу деякого цілого числа N, тобто $N!=1\cdot 2\cdot 3...(N-1)\cdot N$ (число N задається введенням з клавіатури). Записати результат у змінну `A`.
2) Порівняти отриману величину з виразом, отриманим за формулою Стірлінга $\sqrt{2\pi N}\left(\frac{N}{e}\right)^N$. Записати результат у змінну `B`.
3) Використовуючи умовний оператор, порівняти отримані числа. Якщо A = B вивести на екран фразу «Дорівнюють», якщо A > B – «Факторіал більше», A < B – «Факторіал менше».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 5

1) Використовуючи один із циклічних операторів, знайти безпосереднім розрахунком суму N членів гармонічного ряду, тобто $\sum_{k=1}^{N}\frac{1}{k}=\frac{1}{1}+\frac{1}{2}+\frac{1}{3}...+\frac{1}{N}$ (число N задається введенням з клавіатури). Записати результат у змінну `A`.
2) Порівняти отриману величину з виразом $\ln(N)+0.5772$ (формула Ейлера). Записати результат у змінну `B`.
3) Використовуючи умовний оператор, порівняти отримані числа. Якщо A = B вивести на екран фразу «Дорівнюють», якщо A > B – «Сума більша», A < B – «Сума менша».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 6

1) Використовуючи один із циклічних операторів, розрахувати наближене значення функції $\cos(x)$ за формулою Тейлора

$\cos(x)=\sum_i(-1)^i\frac{x^{2i}}{(2i)!}=1-\frac{x^2}{2}+\frac{x^4}{24}-\frac{x^6}{720}+\ldots$ (величина x задається введенням з клавіатури). Сумування проводити поки черговий доданок не стане меншим за 10^{-5}.

2) Використовуючи умовний оператор, порівняти отримані вирази з точним значенням функції $\cos(x)$ (ця функція знаходиться у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`). Вивести на екран відповідне повідомлення.
3) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 7

1) Використовуючи один із циклічних операторів, розрахувати наближене значення функції e^x за формулою Тейлора $e^x=\sum_i\frac{x^i}{i!}=1+x+\frac{x^2}{2}+\frac{x^3}{6}+\ldots$ (величина x задається введенням з клавіатури). Сумування проводити доти, доки черговий доданок не стане меншим за 10^{-6}.
2) Використовуючи умовний оператор, порівняти отримані вирази з точним значенням функції e^x (ця функція знаходиться у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`). Вивести на екран відповідне повідомлення.
3) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 8

1) Використовуючи один із циклічних операторів, розрахувати наближене значення функції $\sin(x)$ за формулою Тейлора $\sin(x)=\sum_i(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!}=x-\frac{x^3}{6}+\frac{x^5}{120}+\ldots$ (величина x задається введенням з клавіатури). Сумування проводити доти, доки черговий доданок не стане меншим за 10^{-5}.
2) Використовуючи умовний оператор, порівняти отримані вирази з точним значенням функції $\sin(x)$ (ця функція знаходиться у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`). Вивести на екран відповідне повідомлення.
3) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 9

1) Використовуючи один із циклічних операторів, розрахувати наближене значення функції $\ln(1+x)$ за формулою Тейлора $\ln(1+x)=\sum_i (-1)^{i+1}\frac{x^i}{i}=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}...$ (величина x задається введенням з клавіатури). Сумування проводити доти, доки черговий доданок не стане меншим за 10^{-4}.
2) Перевірити, що значення x потрапляє в область збіжності ряду Тейлора для функції $\ln(1+x)$. Якщо ні, програма повинна видати повідомлення про помилку.
3) Використовуючи умовний оператор, порівняти отримані вирази з точним значенням натурального логарифму (ця функція знаходиться у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`). Вивести на екран відповідне повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 10

1) Використовуючи один із циклічних операторів, розрахувати наближене значення функції $\mathrm{arctg}(x)$ за формулою Тейлора $\mathrm{arctg}(x)=\sum_i (-1)^{i-1}\frac{x^{2i-1}}{2i-1}=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}...$ (величина x задається введенням з клавіатури). Сумування проводити доти, доки черговий доданок не стане меншим за 10^{-4}.
2) Перевірити, що значення x потрапляє в область збіжності ряду Тейлора для функції арктангенс. Якщо ні, програма повинна видати повідомлення про помилку.
3) Використовуючи умовний оператор, порівняти отримані вирази з точним значенням натурального логарифму (ця функція знаходиться у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`). Вивести на екран відповідне повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 11

1) Нехай задана кількість секунд S, які пройшли з початку доби (величина S задається введенням з клавіатури).
2) Перевірити, що значення S є додатнім та не перевищує кількості секунд в

одній добі. Якщо ні, програма повинна видати повідомлення про помилку.
3) Використовуючи один із циклічних операторів, розрахувати, скільки годин, хвилин та секунд минуло з початку доби.
4) Вивести результат на екран у відповідному форматі («гг:хх:сс»).
5) Напишіть другий варіант програми. Реалізуйте цикл з пункту 3, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 12

1) За заданим номером дня року *R* (від 1 до 365) визначити номер місяця та день місяця (величина *R* задається введенням з клавіатури).
2) За допомогою умовного оператора реалізувати можливість вибору високосного чи невисокосного року.
3) Перевірити, що значення *R* є додатнім та не перевищує кількості днів у році (з урахуванням можливої високосності). Якщо це не так, програма повинна видати повідомлення про помилку.
4) Вивести на екран результат у форматі «дата місяця . номер місяця».
5) Напишіть другий варіант програми. Реалізуйте завдання з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 13

1) Для заданого числа *X* (від 1 до 10 000) визначити всі власні множники (величина *X* задається введенням з клавіатури).
2) Перевірити, що значення *X* знаходиться у вказаному інтервалі. Якщо це не так, програма повинна видати повідомлення про помилку.
3) Послідовно, через кому, вивести на екран власні множники числа *X* в довільному порядку.
4) Напишіть другий варіант програми. Реалізуйте завдання з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 14

1) Для заданого числа *X* (від 1 до 10 000) визначити, чи є воно простим, чи ні (величина *X* задається введенням з клавіатури). Нагадаємо, що простим називається число, яке ділиться без остачі лише на 1 та на саме себе. Використайте метод прямого перебору в циклі.
2) Перевірити, що значення *X* знаходиться у вказаному інтервалі. Якщо це не так, програма повинна видати повідомлення про помилку.
3) Вивести на екран результат аналізу *X* у вигляді повідомлення «Просте число» або «Не просте число».
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1,

використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 15

1) Використовуючи один із циклічних операторів, перевести задане число X (від 0 до 1023) у двійкове представлення (величина X задається введенням з клавіатури).
2) Перевірити, що значення X знаходиться у вказаному інтервалі. Якщо це не так, програма повинна видати повідомлення про помилку.
3) Вивести на екран число X у двійковій формі. Наприклад, числу 423_{10} повинно відповідати 110100111_2, а числу 687_{10} – 1010101111_2.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 16

1) Використовуючи один із циклічних операторів, перевести задане число X, яке задається введенням з клавіатури (від 0 до 1023), у вісімкову систему числення (по основі 8).
2) Перевірити, що значення X знаходиться у вказаному інтервалі. Якщо це не так, програма повинна видати повідомлення про помилку.
3) Вивести на екран число X у вісімковій формі. Наприклад, числу 423_{10} повинно відповідати 647_8, а числу 687_{10} – 1257_8.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 17

1) Використовуючи один із циклічних операторів, розрахувати наближене значення інтегралу від заданої функції за формулою $I = \int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=0}^{N} f(x_i)$, де $x_i = a + \frac{b-a}{N} i$ (N – довільне, достатньо велике ціле число). Покласти $a = 0$, $b = 1$, $f(x) = x^3$.
2) Порахувати наближене значення інтегралу для трьох різних значень кількості доданків N.
3) Використовуючи умовний оператор, порівняти всі отримані вирази з точним значенням інтегралу $I = 1/4$. Вивести на екран відповідні повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 18

1) Використовуючи один із циклічних операторів, розрахувати наближене значення інтегралу від заданої функції за формулою $I = \int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=0}^{N} f(x_i)$, де $x_i = a + \frac{b-a}{N}i$ (N – довільне, достатньо велике ціле число). Покласти $a = 0$, $b = \pi$, $f(x) = \sin(x)$. Функція синус доступна у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`.
2) Порахувати наближене значення інтегралу для трьох різних значень кількості доданків N.
3) Використовуючи умовний оператор, порівняти всі отримані вирази з точним значенням інтегралу $I = 2$. Вивести на екран відповідні повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 19

1) Використовуючи один із циклічних операторів, розрахувати наближене значення інтегралу від заданої функції за формулою $I = \int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=0}^{N} f(x_i)$, де $x_i = a + \frac{b-a}{N}i$ (N – довільне, достатньо велике ціле число). Покласти $a = 1$, $b = 2$, $f(x) = 4x^3$.
2) Порахувати наближене значення інтегралу для трьох різних значень кількості доданків N.
3) Використовуючи умовний оператор, порівняти всі отримані вирази з точним значенням інтегралу $I = 15$. Вивести на екран відповідні повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

Варіант 20

1) Використовуючи один із циклічних операторів, розрахувати наближене значення інтегралу від заданої функції за формулою $I = \int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=0}^{N} f(x_i)$, де $x_i = a + \frac{b-a}{N}i$ (N – довільне, достатньо велике ціле число). Покласти $a = 0$, $b = \pi/2$, $f(x) = \cos(x)$. Функція косинус доступна у бібліотеці, що підключається директивою `#include <math.h>` або `#include <cmath>`.

2) Порахувати наближене значення інтегралу для трьох різних значень кількості доданків N.
3) Використовуючи умовний оператор, порівняти всі отримані вирази з точним значенням інтегралу $I = 1$. Вивести на екран відповідні повідомлення.
4) Напишіть другий варіант програми. Реалізуйте цикл з пункту 1, використовуючи оператор `goto`. Див. приклад з варіанту 1.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Назвіть основні логічні операції та правила їх виконання. Як вони позначаються у мові програмування C/C++?
2. Назвіть основні операції порівняння для інтегральних типів даних. Як операції порівняння позначаються у мові програмування C/C++?
3. Виходячи з власного досвіду програмування, впорядкуйте групи операцій за пріоритетом виконання: арифметичні операції, логічні операції, операції порівняння. Обґрунтуйте свою думку.
4. Які оператори використовуються в C/C++ для розгалуження?
5. Наведіть синтаксис тернарного умовного оператора у C/C++ та поясніть правило його виконання. Чим, на вашу думку, цей оператор може бути кращим або гіршим за умовний оператор `if-else`?
6. Наведіть синтаксис умовного оператора у C/C++ та правило його виконання.
7. Для чого потрібен оператор `switch`? Який синтаксис цього оператора? Наведіть приклад, коли оператор `switch` є більш зручним у використанні, ніж оператор `if-else`.
8. Для чого використовуються цикли у мовах програмування? Які оператори циклів передбачені в мові C/C++?
9. Наведіть синтаксис і поясніть принцип роботи оператора `while` та `do-while`.
10. Наведіть синтаксис і поясніть принцип роботи оператора `for`.
11. Для чого використовується оператори `break` та `continue` у мові програмування C/C++?
12. Синтаксис та приклад застосування оператора `goto`.

Лабораторна робота № 3.
Функціональне програмування на C/C++

Мета роботи:

1) Ознайомитися з поняттям функції та навчитись їх реалізовувати.
2) Опанувати методи функціонального програмування.
3) Навчитись використовувати у функціях аргументи за замовчуванням.
4) Навчитись створювати вбудовані функції.
5) Навчитись перевантажувати функції.

КОНЦЕПЦІЯ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

Функціональне програмування[1] – концепція (парадигма) програмування, згідно якої програма розглядається як набір окремих ***підпрограм***. Кожна з цих підпрограм виконує певну послідовність дій і називається ***функцією*** або ***процедурою***.

У середині та другій половині XX ст. концепція функціонального програмування викликала революцію в галузі інженерії програмного забезпечення. Вона дозволила суттєво спростити розробку будь-яких програм і, в першу чергу, програм великого розміру.

Щоб переконатись в дієвості цієї концепції, припустимо, що необхідно написати програму досить великого об'єму, наприклад, програму, в якій реалізовано певний варіант завдань з усіх лабораторних робіт, наведених у цій книзі. Якщо для цього буде використовуватись спрощений стиль написання програм, який застосовувався раніше, – коли весь код програми записується в тілі єдиної функції `main()` – це призведе до того, що тіло функції `main()` буде містити тисячі рядків коду (лістинг 3.1, розмір блоків коду обрано довільним чином):

Лістинг 3.1:

```
        1:  int main( void )
        2:  {
        3:      // Код для лабораторної 1.
     4-50:      // ...
       51:
       52:      // Код для лабораторної 2.
   53-200:      // ...
      201:
```

[1] Функціональне програмування часто називають ***функціонально-орієнтованим програмуванням***. Цей термін, однак, не є синонімом до *функційного програмування*.

```
     202:      // Код для інших лабораторних.
203-5000:      // ...
    5001:
    5002:      return 0;
    5003:  }
```

Вочевидь, пошук певного фрагменту коду у такій функції `main()`, написання, модифікація, навіть, просто аналіз роботи цього коду будуть суттєво ускладнені через необхідність одночасно «тримати в голові» десятки, а може і сотні рядків коду. Не слід забувати і про надійність написаного коду – в ньому можуть бути достатньо неочевидні «паразитні зв'язки» через змінні, що одночасно використовуються різними блоками коду (наприклад, блоками, що відносяться до різних лабораторних робіт). З точки зору професійного програмування така програма написана жахливо! Її відлагодження та модернізація можуть вимагати на порядок більшої кількості людино-годин, ніж потрібно для програм такого обсягу.

Згідно концепції функціонального програмування правильний підхід до написання великих програм полягає у розділенні коду програми на відносно невеликі за обсягом фрагменти, кожен з яких виконує певну задачу і може бути реалізованим як окрема функція (або процедура). Приклад такого коду наведено в лістингу 3.2.

Лістинг 3.2:

```
 1: int main( void )
 2: {
 3:     Lab01(); // Лабораторна 1.
 4:     Lab02(); // Лабораторна 2.
 5:     Lab03(); // Лабораторна 3.
 6:     Lab04(); // Лабораторна 4.
 7:     Lab05(); // Лабораторна 5.
 8:     Lab06(); // Лабораторна 6.
 9:     Lab07(); // Лабораторна 7.
10:     Lab08(); // Лабораторна 8.
11:     Lab09(); // Лабораторна 9.
12:     Lab10(); // Лабораторна 10.
13:     // ...
14:     return 0;
15: }
```

Як видно з лістингу, тіло функції `main()` має обсяг всього 12 рядків (рядки 3-14). Корисна робота цієї функції зводиться лише до виклику функцій `Lab01()`, … `Lab10()`, які, власне, і виконують завдання лабораторних робіт. Отже, написаний код (рядки 3-12) є однотипним і легким для розуміння.

Для успішної компіляції коду з лістингу 3.2 необхідно належним чином

реалізувати функції `Lab01()`, … `Lab10()`. При цьому процес розбиття конкретної задачі на підзадачі можна продовжувати. Наприклад, задача, яку має виконувати функція `Lab03()`, може бути реалізована за допомогою декількох допоміжних функцій, які будуть викликатись в тілі функції `Lab03()`. Діючи таким чином, можна створити програму, код якої буде записано у вигляді невеликих за об'ємом фрагментів – функцій, з якими зручно працювати.

На практиці розбиття коду програми на окремі функції та процедури також може бути зумовлено необхідністю багаторазового повторення одних і тих самих дій (операцій, розрахунків тощо) для різних значень вхідних параметрів. Щоб уникнути повторень однакового коду при роботі з різними даними, відповідні оператори виділяються в самостійну частину програми, яка оформлюється як функція або процедура. Ця функція або процедура може бути викликана з будь-якої іншої підпрограми для довільного припустимого набору вхідних даних, що в цілому спрощує процес розробки програми. Наприклад, функція `printf()` дозволяє виводити форматоване текстове повідомлення, що значно спрощує роботу з консоллю (достатньо викликати вже готову функцію і не потрібно самостійно реалізовувати низькорівневий код для форматованого виводу повідомлення на консоль).

Ключовими поняттями у функціональному програмуванні є вже згадані вище функції та процедури.

Функція є сукупністю операторів (команд), що виконує певну задачу, і має власне унікальне ім'я. Функції можуть мати ***аргументи*** – параметри, що передаються у функцію під час її виклику. Функції також повертають результат своєї роботи у вигляді значення, пов'язаного з певним типом даних.

З точки зору основ програмування ***процедура*** є майже повним аналогом функції. Проте, на відміну від функції, процедура не повертає результат своєї роботи.

У мові програмування C/C++ і функції і процедури традиційно називаються єдиним терміном – функція. Якщо функція в коді C/C++ повертає результат будь-якого типу, окрім типу `void`, така функція з точки зору загального програмування є звичайною функцією (яка повертає значення). Якщо ж функція в коді C/C++ повертає результат типу `void` (порожній тип), така функція, фактично, є процедурою. Наприклад, функція `main()` з лістингу 3.2 є звичайною функцією (повертає значення типу `int`). Якщо ж функцію `Lab03()` оголосити (про це див. нижче) як `void Lab03( void )`, ця функція з точки зору загального програмування буде процедурою.

Наприкінці ще раз зазначимо, що за винятком того, результат якого типу (`void`, чи не `void`) повертає підпрограма, принципової відмінності між функціями та процедурами у мові C/C++ немає. Саме завдяки цьому в програмах, написаних мовою C/C++, в усіх зазначених випадках користуються єдиним терміном – *функція*, не розрізняючи окремо функції, які повертають значення, та процедури, які не повертають значення.

ФУНКЦІЇ В МОВІ ПРОГРАМУВАННЯ C/C++

У мові C/C++ всі підпрограми називаються ***функціями***.

Функції, що використовуються в програмі, можуть розташовуватися як у .c/.cpp файлі основної програми (наприклад, у файлі, де реалізована функція `main()`), так і в окремому файлі проекту, або у спеціальних файлах-бібліотеках.

Мова C/C++ пропонує програмісту великий набір вбудованих функції, зібраних у стандартних бібліотеках C/C++. Відповідні функції стають доступними при підключенні потрібних файлів заголовків та файлів бібліотек (останнє виконується автоматично в сучасних IDE). Наприклад, для використання стандартних функцій для роботи з консоллю слід включити в програму директиву `#include <stdio.h>` та/або `#include <iostream>`; для роботи з математичними функціями слід включити директиву `#include <math.h>` та/або `#include <cmath>`, тощо.

Визначення та оголошення функції

Перш ніж використовувати функцію в програмах на C/C++, її слід ***визначити*** (define). ***Визначення функції*** полягає у написанні її коду – набору операторів, що складають ***тіло функції***, а також у вказуванні її специфічних рис: імені, можливого набору параметрів, типу результату, що має повертати функція, тощо. Спрощений вигляд визначення функції наступний:

```
Тип_результату Ім'я_функції( Параметри )
{
    Тіло_функції (оператори);
}
```

Рядок `Тип_результату Ім'я_функції( Параметри )` називається ***заголовком функції***.

`Тип_результату`, визначає тип того значення, яке повертає дана функція. Цей тип може бути одним із стандартних інтегральних типів C/C++ (`int`, `double`, `char` тощо), одним із нових типів, введених програмістом (це питання буде розглядатись у наступних лабораторних роботах), або типом `void`.

`Ім'я_функції` – унікальний ідентифікатор, який дозволяє звертатись до даної функції у будь-якій точці програми.

`Параметри` функції – тимчасові змінні, які доступні всередині функції і використовуються для передавання даних із точки виклику функції в саму цю функцію. Список параметрів функції вказується у круглих дужках після її імені і має вигляд:

```
( Тип1 Зм1, Тип2 Зм2, ..., ТипN ЗмN ),
```

де Зм1, Зм2, ..., ЗмN – назви змінних, Тип1, Тип2, …, ТипN – відповідно їх типи. Якщо функція не має параметрів, то після її імені вказуються просто пусті круглі дужки `()` або в цих дужках зазначається `void`.

`Тіло функції` містить набір операторів, які в сукупності реалізують певну задачу. Тіло функції завжди включається в фігурні дужки `{}`.

Розглянемо декілька прикладів визначення функції. Функція `main()` у лістингу 1 (див. рядок 5) має ім'я `main`, повертає результат типу `int` і не має параметрів, на що вказує тип `void`, зазначений в круглих дужках `( void )`. У лістингу 3.3 наведено ще два простих приклади визначення функцій:

Лістинг 3.3:
```
 1: double Sqr( double x ){
 2:     return x*x;
 3: }
 4:
 5: void PrintError()
 6: {
 7:     printf("Error!!!\n");
 8: }
```

Перша функція має ім'я `Sqr()`. Вона має один параметр типу `double` з ім'ям `x` і повертає результат типу `double`. Друга функція має ім'я `PrintError()`. Вона не має параметрів і нічого не повертає (тип результату `void`).

Окрім визначення, кожну функцію ще можна ***оголосити*** (declare). ***Оголошення функції*** полягає у записі її заголовку, після чого ставиться кома з крапкою `;`, як після оператора. Записане таким чином оголошення називається ***прототипом функції***. Наприклад, для функцій з лістингу 3.3, прототипи мають вигляд:

```
double Sqr( double x );
void PrintError();
```

Написання прототипу функції не є обов'язковим. Більше того, якщо функція визначена в тексті програми раніше, ніж вона викликається, її визначення буде одночасно сприйматись і як її оголошення.

Оголошення функції заздалегідь інформує компілятор про її специфічні риси (ім'я, тип результату, параметри тощо), що іноді може бути корисно. Наприклад, припустимо, що була створена програма з лістингу 3.2, в якій використовуються функції `Lab01()`, … `Lab10()`. Аби не шукати описи цих функцій всередині .c/.cpp файлу іноді буває доцільно навести прототипи цих

функцій (можливо з детальними коментарями) на початку програми, після директив `#include <...>`, щоб відразу ж дати програмісту (та компілятору) інформацію щодо ключових функцій програми.

Іншим прикладом є використання функцій, визначених в одному .c/.cpp файлі, які викликаються в інших .c/.cpp файлах. Щоб при такому виклику не виникало помилок, необхідно проінформувати компілятор, що деяка функція, наприклад, `int f( int x )`, реалізована не в тому .c/.cpp файлі, де вона викликається, а в іншому. Щоб це зробити, в тому файлі, де викликається функція `f()`, записується її прототип `extern int f( int x );` із ключовим словом `extern`. Слово `extern` вказує компілятору на те, що функція визначена поза межами поточного .c/.cpp файлу. Зазначимо, що саме такий підхід використовується для функцій зі стандартної бібліотеки C/C++. Наприклад, у файлі stdio.h, який підключається до програми директивою `#include <stdio.h>` міститься прототип функції `printf()` з `extern`, що дозволяє вільно використовувати цю функцію у власних програмах на C/C++.

Тіло функції. Локальні та глобальні змінні. Завершення роботи функції

Тіло функції обмежене фігурними дужками `{}` і являє собою послідовність операторів. Всередині тіла функції (часто кажуть просто всередині функції) можуть оголошуватися свої, внутрішні змінні, які називаються ***локальними***. Областю дії локальної змінної є той блок операторів (всередині тіла функції), де ця змінна оголошена. Час життя локальної змінної відповідає часу життя того блоку операторів, в якому вона оголошена.

У лістингу 3.4 наведено текст програми з функцією `main()`, в тілі якої вводяться три локальні змінні: `int i`, `int S` та `int j`. Змінні `i` та `S`, оголошені в рядках 7 та 8, мають область дії – тіло функції `main()` і час життя, що відповідає часу виконання `main()`. Локальна змінна `j` визначена лише всередині циклу `for( int j = 1; j < i; j++ )` у рядку 10. Її час життя відповідає часу виконання цього оператора `for`, а область дії – тілу цього оператора (рядки 12-13). Використання змінної `j` поза межами вказаного циклу `for`, наприклад, якщо у рядку 16 замінити `S` на `j`, призведе до помилки компіляції програми.

Лістинг 3.4:

```
 1: #include <stdio.h>
 2:
 3: int gS = 0; // Глобальна змінна.
 4:
 5: int main( void )
 6: {
```

```
 7:      int i = 10; // Локальні змінні.
 8:      int S = 0;
 9:
10:      for( int j = 1; j < i; j++ )
11:      {
12:          S += j;
13:          gS += j*j;
14:      }
15:
16:      printf("Sum(1, 2, .. 9) = %d\n", S);
17:      printf("Sum(1*1, 2*2, .., 9*9) = %d\n", gS);
18:
19:      return gS - S;
20:  }
```

Окрім локальних змінних, у функціях можна використовувати ***глобальні змінні***, оголошені поза межами будь-якої функції, зокрема, й функції `main()`. Глобальні змінні мають час життя, що співпадає з часом виконання всієї програми і область дії, що за замовчуванням відповідає .c/.cpp файлу, в якому визначена глобальна змінна. Наприклад, змінна `int gS`, оголошена в рядку 3 лістингу 3.4 є глобальною. Її можна використовувати в будь-якій функції, що знаходиться в цьому ж самому .c/.cpp файлі.

Більше того, глобальні змінні можуть бути доступні і в інших .c/.cpp файлах. Наприклад, нехай код з лістингу 3.4 збережено у файлі 1.cpp, а доступ до змінної `gS` необхідно отримати з іншого файлу 2.cpp. Якщо в файлі 2.cpp оголосити змінну `gS` з ключовим словом `extern` як `extern int gS;`, всі функції в 2.cpp отримають доступ до глобальної змінної `gS`, оголошеної в файлі 1.cpp. Цей приклад показує, що можна, використовуючи ключове слово `extern`, розділяти між декількома .c/.cpp файлами не тільки код (функції), але й глобальні змінні.

Якщо функція має повертати результат будь-якого типу окрім `void`, вона повинна містити один або декілька операторів `return`, розглянутих у лабораторній роботі 2. В цьому випадку виконання оператора `return` з деяким операндом *X* приводить до завершення роботи функції з одночасним поверненням нею значення *X*. Це значення має відповідати тому типу даних, що зазначений в заголовку функції. Наприклад, у лістингу 3.4 функція `main()` має повертати значення типу `int` і вона, дійсно, повертає величину `gS - S`, яка має тип `int`.

Якщо функція повертає результат типу `void`, запис для неї оператора `return` без операнда не є обов'язковим. В цьому випадку вважається, що робота функції автоматично завершується при виконанні всіх операторів, присутніх в її тілі.

Наведемо приклади невірного або недоречного використання оператора

`return` при написанні функцій:

Лістинг 3.5:

```
 1: int f1()
 2: {
 3:     return 0.0;
 4: }
 5:
 6: int f2()
 7: {
 8:     return;
 9: }
10:
11: void f3()
12: {
13:     return 0;
14: }
15:
16: void f4()
17: {
18:     return;
19:     printf("Hello!\n");
20: }
```

У функції `int f1()` оператор `return 0.0;` повертає значення 0.0 (тобто число типу `double`), яке не відповідає значенню типу `int`, що має повертати функція. Однак, зворотний випадок, коли функція `double f5()` за допомогою `return 0;` повертає значення 0, є цілком припустимим, оскільки значення 0 типу `int` може бути автоматично конвертоване у тип `double` (буде відбуватись розширення типу `int` до типу `double`).

У функції `int f2()` помилково використовується оператор `return;` без операнда, хоча функція має повертати число типу `int`.

Функція `void f3()` не повинна нічого повертати, але містить оператор `return 0;` із зайвим операндом.

Нарешті, функція `void f4()` містить правильно записаний оператор `return;` у рядку 18. Проте вже наступний рядок 19, що містить `printf("Hello!\n");` виконуватись не буде, оскільки оператор `return` у рядку 18 перериває виконання функції. Відповідно, код рядку 19, хоч сам по собі він записаний вірно, виявляється недосяжним (unreachable code), про що буде виведено попередження під час компіляції програми. Створюючи програми слід пам'ятати про цю особливість – оператори, записані після `return`, виконуватися не будуть.

Наприкінці зазначимо, що деякі сучасні IDE можуть за необхідності

автоматично вставляти в код функції оператор `return` з деяким значенням потрібного типу, якщо відповідний рядок не написаний програмістом (наприклад, див. 6, де наведено код програми без `return`). По можливості бажано уникати таких ситуацій, оскільки явний запис в тілі функції оператора `return` вважається гарним стилем програмування і дозволяє писати більш зрозумілі і надійні програми.

Виклик функції. Передача параметрів у функцію

Виклик функції у мові C/C++ реалізується за допомогою ***оператора виклику функції***, який позначається круглими дужками `()` після імені функції. Синтаксис цього оператора наступний:

```
z = f1(Арг1, ..., АргN);
```

або

```
f2(Арг1, ..., АргN);
```

Тут `Арг1`, ..., `АргN` – аргументи, які передаються в функцію `f1()` або `f2()`. Значення цих аргументів повинні відповідати тим типам параметрів, які мають приймати функції. Порядок аргументів має значення. Наприклад, якщо функція `f1()` визначена як `double f1( double x, int i )`, а її викликати як `f1(0, 1.0)`, це буде сприйняте компілятором як помилка. Помилка в тому, що другий аргумент `1.0` неможливо автоматично конвертувати в тип `int` (хоча перший аргумент без проблем конвертується з типу `int` у тип `double`).

Згідно наведеного синтаксису функція `f1()` має повертати певне значення, яке буде збережене в змінній `z`. Функція `f2()` або нічого не повертає, або те значення, яке вона повертає, ігнорується.

У мові C/C++ параметри в функцію передаються ***за значенням***. Це означає, що при виклику функції їй будуть передані копії значень усіх вказаних параметрів-змінних, але не самі відповідні змінні. Отже, модифікація значень вхідних параметрів функції ніяк не вплине на відповідні змінні. Реалізувати доступ до цих змінних можна за допомогою вказівників або посилань, які будуть розглядатись в наступних лабораторних роботах[1].

Ще одна особливість використання функцій полягає в тому, що їх

[1] Зазначимо, що в сучасних IDE, якщо програміст намагається модифікувати змінну, яка передається за значенням у функцію, це іноді може сприйматись компілятором як натяк на те, що необхідно забезпечити доступ до цієї змінної. В цьому випадку компілятор може приховано від програміста генерувати код, в якому вже використовуються вказівники або посилання і відповідна змінна вже передається у функцію не за значенням, а за адресою.

параметри, які описані в заголовку функції і використовуються в її тілі, називаються ***формальними***. Формальні параметри є деякою абстракцією, вони не мають певних конкретних значень. Наприклад, параметр `x` функції `double Sqr( double x )` з лістингу 3.3 може бути довільним числом типу `double`. В момент, коли пишеться код цієї функції точне значення параметра `x`, саме для якого буде викликана функція, наперед невідоме. Ті ж самі параметри, що вказуються під час виклику функції є ***фактичними***. Коли функція викликається, всі її параметри мають бути явно задані у вигляді конкретних значень потрібного типу.

Отже, у наведених вище описах параметри функцій `Зм1`, `Зм2`, ..., `ЗмN` є формальними, а `Арг1, ..., АргN` – фактичними. Під час виклику функції фактичні параметри, вказані в операторі виклику функції `()`, передаються у функцію – підставляються замість її формальних параметрів. Вже при виконанні тіла функції передані їй параметри виконують роль локальних змінних.

Функції з аргументами за замовчуванням

Один або декілька параметрів функції можна пов'язати з ***аргументами за замовчуванням***, присвоївши їм певні, наперед визначені, значення. Такі аргументи програміст може не зазначати при виклику функції – в цьому випадку замість «пропущеного» аргументу функції буде автоматично передане вказане значення за замовчуванням.

Аргументи за замовчуванням можуть спростити код програми і є зручними у використанні. Вони мають зазначатись в прототипі функції (ось ще один приклад того, чому важливо писати прототипи функцій!). Аргументи за замовчуванням мають визначатись послідовно, починаючи від самого останнього аргументу функції і закінчуючи першим.

Лістинг 3.6:

```
 1: #include <stdio.h>
 2:
 3: int Sum( int a, int b, int c = 0 );
 4:
 5: int Sum( int a, int b, int c )
 6: {
 7:     return a + b + c;
 8: }
 9:
10: int main( void )
11: {
12:     Sum(1, 1);
13:     Sum(1, 1, 2);
```

```
14:
15:       return 0;
16:  }
```

У лістингу 3.6 визначається прототип функції `int Sum( int a, int b, int c = 0 )`, де вказується, що її останній аргумент за замовчуванням має значення 0. Відповідно виклик функції `Sum(1, 1);` в рядку 12 приводить до того, що їй будуть передаватись аргументи a = 1, b = 1 та аргумент за замовчуванням c = 0. У наступному рядку 13 реалізований звичайний виклик функції `Sum(1, 1, 2);`, де явно задаються всі три аргументи, а аргумент за замовчуванням не використовується.

Вбудовані (inline) функції

Введення функцій суттєво спрощує розробку складних програм, полегшує їх модифікацію, дозволяє змінювати алгоритм тої, чи іншої функції лише в місці її опису. Проте, разом з тим, кожен виклик функції потребує певних витрат пам'яті та процесорного часу.

Під час виклику функції необхідно скопіювати в тимчасову область пам'яті – *стек* – значення всіх параметрів, які передаються функції. Потім потрібно передати управління коду функції, а після завершення її роботи слід видалити зі стеку раніше записані в нього параметри. Всі ці операції, пов'язані із викликом функції, можуть виконуватись протягом кількох (іноді десятків) тактів процесора і можуть вимагати використання достатньо великого об'єму пам'яті. Якщо одночасно з цим, код самої функції виконується досить швидко (за кілька тактів процесора), «коефіцієнт корисної дії» оператора виклику функції виявляється досить низьким. Наприклад, якщо для виклику функції слід витрати 4 такти, а код самої функції виконується за 3 такти, корисна робота при виклику функції буде виконуватись лише протягом 100% × 3 / (3 + 4) = 43% часу, потрібного для виклику та роботи функції. В цьому або подібних випадках використання оператора виклику функції `()`, формально, призводить до уповільнення роботи програми, порівняно з випадком, коли код тіла функції безпосереднього записується в тексті програми як набір операторів.

Вказана особливість оператора виклику функції може бути важлива для *малих* функцій, код яких містить лише невелику кількість операторів (типово менше 5). Для достатньо великих функцій, обсягом 10 і більше рядків, витрати на виклик функції, як правило, виявляються несуттєвими, порівняно з витратами на виконання тіла функції.

Крім того, якщо мала функція викликається в тексті програми лише кілька (< 10) разів, додаткові витрати процесорного часу та пам'яті, пов'язані з оператором виклику функції, зазвичай, теж несуттєві. Однак, якщо ця функція викликається в програмі багато разів (наприклад, у циклі), відповідні витрати

можуть стати досить значними.

Для того, щоб зберегти розбиття коду на функції і в той же час пришвидшити роботу з малими функціями, у мові C++ використовується механізм вбудованих функцій. ***Вбудована (inline) функція*** – це функція, код якої безпосередньо вставляється компілятором в машинний код програми без використання оператора виклику функції. З точки зору програміста, вбудована функція – звичайна функція, виклик якої в тексті програми здійснюється за тими ж правилами, що і будь-якої іншої функції. Проте при генерації машинного коду, компілятор не генерує код виклику цієї функції, а безпосередньо вставляє в машинний код програми її тіло.

Для позначення функції як вбудованої використовується ***модифікатор функції*** – в даному випадку модифікатор `inline`, який записується перед типом результату, що повертає функція – див. рядок 3 у лістингу 3.7.

Лістинг 3.7:

```
 1: #include <stdio.h>
 2:
 3: inline int Sum( int a, int b, int c )
 4: {
 5:     return a + b + c;
 6: }
 7:
 8: int main( void )
 9: {
10:     int S = 0;
11:
12:     for( int i = 0; i < 30000; i++ )
13:     {
14:         S += i+i+i;
15:     }
16:
17:     for( int i = 0; i < 30000; i++ )
18:     {
19:         S += Sum(i,i,i);
20:     }
21:
22:     return S;
23: }
```

У лістингу 3.7 реалізована вбудована функція `int Sum( int a, int b, int c )`, яка обчислює суму трьох переданих їй чисел (рядки 3-6). Ця функція далі викликається в рядку 19 всередині циклу `for`. Як видно, з цього прикладу виклик вбудованої функції в тексті програми синтаксично ніяк не відрізняється від виклику звичайної (не `inline`) функції.

Якщо функція `Sum()` визначена з ключовим словом `inline`, код тіла функції (рядок 5) буде підставлятись компілятором в той рядок, де вона викликається (рядок 19). В результаті машинний код рядків 19, де викликається вбудована функція (оператор `return` в рядку 5 при цьому пропускається), та 14, де виконується звичайне обчислення суми `S += i+i+i;`, буде ідентичним. Однак, якщо прибрати модифікатор `inline` у рядку 3, у рядку 19 компілятор буде генерувати звичайний виклик функції, що призведе до уповільнення виконання тіла другого циклу `for`. Для 30 тисяч ітерацій циклу, вказаних у лістингу 3.7, затримка у виконанні другого циклу `for` може сягати десятків тисяч тактів процесора.

Використання inline-функцій має певну специфіку.

По-перше, не кожна функція може бути реалізована вбудованою. Компілятор C/C++, зазвичай, не дозволяє[1] робити вбудованою ту функцію, яка містить оператори `while`, `do-while`, `switch`, `for`, `goto`, яка містить вбудований код асемблера або в своєму тілі викликає саму ж себе (тобто використовує рекурсію), має список аргументів змінної довжини, тощо. Якщо компілятор зустрічає функцію, яка оголошена як вбудована, але яка не задовольняє вимогам до вбудованих функцій, він ігнорує модифікатор функції `inline` і реалізує машинний код цієї функції як звичайної функції, яка має викликатись через оператор виклику функції. З іншого боку, особливо під час побудови фінальної (release) версії програми, компілятор може самостійно, навіть без вказаного програмістом ключового слова `inline`, деякі прості функції реалізовувати як вбудовані.

По-друге, компілятор розглядає функцію як вбудовану тільки якщо вона визначена з модифікатором `inline` ще до моменту її виклику в тексті програми. Якщо компілятор зустрічає виклик функції і лише потім в тексті зазначається, що ця функція має бути вбудованою, він генерує звичайний виклик функції і не використовує механізм вбудованих функцій. Наприклад, у лістингу 3.8 функція `f1()` визначається в рядку 1 як вбудована, ще до моменту її виклику у рядку 5. Для цієї функції компілятор у рядку 5 використовує підстановку коду функції, тобто використовує механізм вбудованих функцій. У той же час, функція `f2()` зазначається вбудованою у рядку 8, хоча викликається раніше – у рядку 5. Тому для цієї функції компілятор не використовує механізм вбудованих функцій, а генерує звичайний виклик функції.

Лістинг 3.8:

```
1: inline int f1( int a ){ return a + 1; }
2:
3: int f( int a, int b )
4: {
```

[1] Ці правила можуть відрізнятись для різних стандартів C++, різних компіляторів або навіть для різних версій одного і того ж самого компілятора.

```
5:      return f1(a) + f2(b);
6:  }
7:
8:  inline int f2( int b ){ return b + 2; }
```

Таким чином, щоб успішно скористатись механізмом вбудованих функцій, їх зручно розміщувати або на початку файлу програми (як функцію `f1()` у лістингу 3.8), або у файлу заголовку, який включають за допомогою `#include` на початку програми.

Оскільки код вбудованої функції записується безпосередньо в машинний код програми, це може приводити до помітного збільшення об'єму кінцевого файлу, наприклад, .exe файлу застосунку. Іноді це збільшення об'єму коду може бути досить значним, наприклад, від 100 Кб до 2 – 3 Мб, тобто в 20–30 разів.

Перевантаження (overloading) функцій

Перевантаження функцій (function overloading) – це технологія програмування, що дозволяє визначати в тексті програми декілька функцій з одним і тим самим ім'ям, але з різними параметрами.

Такі однойменні функції можуть бути корисні, якщо необхідно зробити декілька однотипних дій, результат яких буде залежати від типу або кількості аргументів. Наприклад, припустимо, що потрібно задати функцію `Sum(int i, int j …)`, яка буде повертати суму своїх аргументів для випадку двох, трьох або чотирьох параметрів[1]. Звичайно ж можна визначити декілька функцій зі схожими назвами та різною кількістю параметрів, наприклад, створити функції з іменами `Sum2()`, `Sum3()`, `Sum4()`, які мають відповідно два, три або чотири аргументи типу `int`, проте такий спосіб розв'язання задачі створює три різні функції замість однієї, що не є елегантним. Недолік такого способу стає очевидним, якщо ще додатково розглянути можливість додавання за допомогою `Sum()` двох, трьох або чотирьох величин типу `float` та `double`. У цьому випадку, якщо не використовувати технологію перевантаження функції, слід було б визначати функції, які працюють з аргументами різного типу, а, отже, мають різне ім'я. Наприклад, функція додавання двох цілих чисел могла б мати ім'я `Sum2i()`, а функції, які виконують аналогічну операцію для чисел типу `float` та `double`, – імена `Sum2f()` та `Sum2d()`.

[1] Для виконання саме цієї задачі можна використати функцію з аргументами за замовчуванням. Можна створити функцію `Sum(int i, int j, int k = 0, int l = 0)`, в якій задати два її останні параметри за замовчуванням рівними нулю. Проте в загальному випадку, для довільної задачі, такий підхід вже не спрацює, оскільки при виклику функції потрібно буде явно враховувати значення всіх параметрів, які передаються цій функції, не замінюючи їх на деякі значення за замовчуванням.

Перевантаження функцій дозволяє уникнути зазначеної проблеми «розмноження» імен функцій. Мова C++ дозволяє програмісту визначати довільну кількість функцій, які мають однакове ім'я і відрізняються хоча б одним параметром (його типом та/або положенням у списку параметрів функції). Завдяки цьому всі згадані вище функції, що обчислюють суму двох, трьох або чотирьох величин типу `int`, `float`, `double` можуть мати єдине ім'я, наприклад, `Sum()`. Фрагмент коду з визначенням деяких з цих перевантажених функцій наведено у лістингу 3.9.

Лістинг 3.9:

```
 1: int Sum( int a, int b )
 2: { return a + b; }
 3:
 4: int Sum( int a, int b, int c )
 5: { return a + b + c; }
 6:
 7: int Sum( int a, int b, int c, int d )
 8: { return a + b + c + d; }
 9:
10: float Sum( float a, float b )
11: { return a + b; }
12:
13: float Sum( float a, float b, float c )
14: { return a + b + c; }
15:
16: double Sum( double a, double b )
17: { return a + b; }
18:
19: double Sum( double a, double b, double c )
20: { return a + b + c; }
```

Секрет технології перевантаження функції полягає в тому, що кожній функції в програмі компілятор C++ присвоює унікальне внутрішнє, т. зв. ***декороване (encoded) ім'я***, саме яке і використовується при генерації машинного коду. Це декороване ім'я включає в себе як власне ім'я функції, написане програмістом, так і закодовану за певними правилами кількість та тип параметрів функції. Відповідно, навіть якщо функції мають однакове власне ім'я, але відрізняються типом чи порядком своїх аргументів, вони будуть мати різні декоровані імена, за якими їх можна розрізняти.

Коли перевантажена функція викликається в певному місці програми, компілятор C++ на підставі переданих функції параметрів визначає яку саме перевантажену функцію слід викликати і звертається до неї через відповідне декороване ім'я. При цьому компілятор намагається досягти повного співпадіння типу та порядку параметрів, що передаються функції, з тими

формальними параметрами, які вона має приймати. Якщо ж точного співпадіння не знайдено, компілятор буде намагатись виконати перетворення типу заданих фактичних параметрів: параметр типу `char` може перетворюватись в параметр типу `int`, а параметр типу `float` – у параметр типу `double` тощо. Наприклад, якщо описані перевантажені функції `void print(double x)` та `void print(int x)`, то при виклику `print('b')` буде викликатись друга версія функції, в яку буде передано як параметр числовий код літери `'b'` (число типу `int`).

Завдання

1) Напишіть програму з функціями `Lab01()` та `Lab02()`, всередині яких наведіть код для вашого варіанту раніше виконаних лабораторних робіт 1 та 2. Тіло функції `main()` має містити лише виклики функції `Lab01()`, `Lab02()` та оператор `return`, подібно до того, як це зроблено в лістингу 3.2.
2) Напишіть функцію `Lab03()`, в середині якої реалізуйте завдання пп. 3-6. Додайте виклик функції `Lab03()` в функцію `main()`, подібно до того, як це зроблено в лістингу 3.2.
3) Скориставшись математичною бібліотекою C/C++ (підключивши за допомогою `#include` файли math.h або cmath), напишіть функцію, яка обчислює вказаний математичний вираз для довільного припустимого значення параметра *x*, який має вводитись з клавіатури. За необхідності проведіть перевірку параметра *x* на його відповідність області визначення виразу. Якщо введений параметр *x* не відповідає цій області визначення, програма має виводити повідомлення про невірно введені дані.

Варіант	Вираз	Варіант	Вираз
1	$x^4 - x\sin(x)$	11	$x - x^3\sin(x)$
2	$x^2 + \cos(x)$	12	$x^2\cos(x) - x\cos^2(x)$
3	$e^x\sin(x) - \ln x$	13	$e^x - x^e$
4	x^x	14	$3^{\sin(x)} - \sin^3(x)$
5	$\sin^4(x) + x\cos(x)$	15	$\sin(x) + \ln(x)\cos(x/2)$
6	$2^x - x^2$	16	$e^{-2x} + e^{x/2} + e^{3x/2}$
7	$x^2/(x^2+1)$	17	$\mathrm{ctg}(x) + x\,\mathrm{tg}(x)$
8	$\sin(x)\cos(x)\,\mathrm{tg}(x)$	18	$e^{-x}\,\mathrm{tg}(x)$
9	$3^x - x^3$	19	$x^3 + x^2 + x + 1 - x^{-1}$

Варіант	Вираз	Варіант	Вираз
10	$1-x^2\,\mathrm{tg}(x)$	20	$\sin(4x)+x^4\cos(x/4)$

4) Написати перевантажені функції з п. 3, які мають працювати з типами `int`, `float` та `double`.
5) Написати вбудовані функції `Sqr(x)` та `Cube(x)`, які обчислюють квадрат та куб деякої величини *x* типу `double`. Реалізувати ці функції з аргументом за замовчуванням, що дорівнює нулю.
6) Використовуючи цикл, вивести на консоль інформацію у вигляді 10 рядків даних. У кожному рядку зазначити деяке значення $x = x_0 + (i-1)/10$ (i – номер рядка, $i = \overline{1,10}$) та розраховані для вказаного x значення y та $y^3 - y^2$, де y – значення виразу з п. 3. Обчислення виразу $y^3 - y^2$ провести за допомогою раніше написаних вбудованих функцій з п. 5. Значення x_0, починаючи з якого має виводитись інформація, має дорівнювати номеру варіанту поділеному на 10.

Контрольні запитання

1. Що таке функціональне програмування? В чому його перевага?
2. Що таке функції та процедури? Як реалізуються функції та процедури у мові C/C++?
3. Який синтаксис запису функції в мові C/C++? Наведіть приклади.
4. В чому відмінність між визначенням та оголошенням функції? Що таке прототип функції? Наведіть приклади визначення та оголошення функцій. Коли запис прототипів функцій може бути корисним?
5. Чим відрізняються формальні та фактичні параметри, що передаються функції?
6. Поясніть, що означає твердження, що в мові C/C++ параметри функції передаються за значенням?
7. Чим відрізняються між собою локальні та глобальні змінні однакового типу? Параметри, які передаються функції, є локальними, чи глобальними змінними?
8. Що таке функції з аргументами за замовчуванням? Наведіть приклади реалізації та використання таких функцій.
9. Що таке вбудовані функції? Наведіть приклади реалізації та використання таких функцій.
10. Що таке перевантажені функції? Наведіть приклади реалізації та використання таких функцій.

Лабораторна робота № 4.
Масиви та текстові рядки

Мета роботи:

1) Навчитись створювати та використовувати одновимірні масиви даних.
2) Навчитись створювати та використовувати багатовимірні масиви даних.
3) Навчитись створювати та використовувати текстові рядки в форматі C як масиви символів.
4) Навчитись виконувати сортування, модифікацію, пошук даних у масивах та текстових рядках.

Масиви

Створення одновимірних масивів та доступ до їх елементів

Масив – це впорядкований набір однотипних елементів, з яким пов'язане певне унікальне власне ім'я (ідентифікатор).

Всі елементи масиву розташовуються послідовно в єдиній неперервній області пам'яті. Масив є аналогом звичайної змінної – іменованої області пам'яті, здатної зберігати дані. Однак, на відміну від звичайної змінної, масив може містити не один, а кілька блоків даних однакового типу. Ці блоки однотипних даних називаються ***елементами масиву***.

Кожен елемент масиву має один і той же тип даних, отже, всі елементи мають однаковий розмір у байтах. Кожен елемент у масиві характеризується ***індексом*** – унікальним номером елементу в масиві. Нумерація індексів елементів в масивах завжди починається з нуля: перший елемент масиву має індекс 0, другий елемент – індекс 1, третій – індекс 2 тощо. Для того, щоб звернутись, до будь-якого елементу масиву необхідно знайти ім'я масиву та індекс потрібного елемента.

Синтаксис оголошення масиву у мові C/C++ наступний:

```
Тип_даних Ім'я_масиву[ Розмір_масиву ];
```

`Тип_даних` визначає тип тих даних, які будуть зберігатись в масиві. Цей тип може бути одним із стандартних інтегральних типів C/C++ (`int`, `double`, `char` тощо) або одним із нових типів даних, введених програмістом (це питання буде розглядатись у наступних лабораторних роботах).

`Ім'я_масиву` – унікальний ідентифікатор, який дозволяє звертатись до масиву в цілому або до будь-яких його елементів.

`Розмір_масиву` – натуральне число (≥ 1), що визначає кількість

елементів у масиві. Після оголошення масиву, кількість його елементів змінюватись не може. Такі масиви часто називають ***статичними***. Зазначимо також, що оскільки нумерація індексів елементів в масиві починається з нуля, перший елемент масиву буде мати індекс 0, а останній елемент матиме індекс Розмір_масиву – 1.

Для звернення до певного елементу масиву з індексом Індекс_елемента, використовується конструкція Ім'я_масиву[Індекс_елемента]. Ця конструкція сприймається компілятором як змінна того ж типу, що й тип елементів масиву. Використовуючи форму запису Ім'я_масиву[Індекс_елемента] можна працювати з елементами масиву як зі звичайними змінними: записувати дані в ці змінні, модифікувати ці дані, зчитувати раніше збережені дані тощо. Приклад виконання таких операцій наведено у лістингу 4.1.

Лістинг 4.1:

```
 1: #include <stdio.h>
 2:
 3: int ar[20];
 4:
 5: int main( void )
 6: {
 7:     for( int i = 0; i < 20; i++ ) ar[i] = i;
 8:
 9:     for( int i = 0; i < 20; i++ ) ar[i] += i;
10:
11:     for( int i = 0; i < 20; i++ )
12:         printf("array[%d] = %d\n", i, ar[i]);
13:
14:     return 0;
15: }
```

У рядку 3 лістингу 4.1 оголошується масив з іменем `ar`, здатний зберігати 20 елементів типу `int`. В циклі `for` у рядку 7 всі елементи цього масиву ініціалізуються значенням, що дорівнює індексу елемента в масиві: елемент з індексом 0 буде містити число 0, елемент з індексом 1 буде містити число 1, тощо. Це відбувається за допомогою конструкції `ar[i] = i;`, в якій `ar[i]` є «ім'ям» елемента масиву з індексом `i`. Як видно, робота з довільним елементом масиву нагадує роботу зі змінною того ж самого типу.

У рядку 9, в тілі іншого циклу `for`, відбувається модифікація значень елементів у масиві за допомогою оператора `ar[i] += i;`. Після виконання циклу в `ar[0]` буде вже записане значення 0, в `ar[1]` – значення 2, в `ar[2]` – значення 4 тощо.

Нарешті, в останньому циклі `for` (рядки 11-12) значення елементів `ar[i]` використовуються в функції `printf()` для послідовного виведення

вмісту масиву на консоль.

З лістингу 4.1 очевидно, що масиви дозволяють спростити використання в програмах великої кількості однотипних змінних. Наприклад, зникає потреба придумувати для кожної такої змінної своє власне унікальне ім'я – ідентифікатор. Масиви дозволяють працювати з усіма цими однотипними змінними користуючись тільки єдиним ідентифікатором – ім'ям масиву та індексом потрібного елемента в масиві.

Операції з масивами

Масиви є аналогами змінних. Проте *не всі операції*, дозволені для звичайних змінних, *є дозволеними для масивів*.

Подібно до звичайних змінних, масиви можна ініціалізувати в момент їх оголошення. При цьому, фактично, відбувається *визначення* масиву:

```
Тип_даних Ім'я_масиву[ Розмір_масиву ] = { Зн_1, Зн_2,
…, Зн_останнє };
```

Відповідний список значень `Зн_1`, `Зн_2`, …, `Зн_останнє` має записуватись в фігурних дужках {}. `Зн_1` відповідає елементу з індексом 0, `Зн_2` використовується для ініціалізації елемента з індексом 1, а `Зн_останнє` – для останнього елемента в масиві, з індексом Розмір_масиву – 1. Може застосовуватись і спрощена форма запису

```
Тип_даних Ім'я_масиву[ Розмір_масиву ] = { Значення };
```

коли всі елементи масиву ініціалізуються одним єдиним `Значенням`.

Якщо у списку ініціалізації `{ Зн_1, Зн_2, …, Зн_останнє }` наведено менше значень, ніж кількість елементів в масиві, останні елементи масиву, для яких «не вистачає» значень у списку ініціалізації, будуть ініціалізовані деяким значенням за замовчуванням (зазвичай, це нуль).

Нижче показано декілька прикладів ініціалізації масивів:

```
int ar[20] = { 1, 3, 5, 7 };
int x[10] = {20, 2, 5, 89, 7, 5, 3, 6, 1, 4};
int y[10] = { -1 };
```

Масив `ar` буде містити чотири перші елементи, встановлені у значення 1, 3, 5, 7. Всі інші елементи цього масиву будуть містити значення за замовчуванням. У масиві `x` всі елементи будуть ініціалізовані значеннями, заданими за допомогою списку ініціалізації: наприклад, перший елемент `x[0]` буде містити 20, а шостий елемент `x[5]` буде містити 5. Всі елементи останнього

масиву y будуть ініціалізовані значенням –1.

На відміну від звичайних змінних, безпосередні дії над масивами як над цілісними блоками даних не дозволяються[1]. Обробка масивів має проводитись поелементно. До окремих елементів масивів можна застосовувати стандартні операції та функції, якщо вони дозволені для змінних того ж самого типу, проте виконання відповідних операцій та функцій для всього масиву може бути не дозволеним. Наприклад, елементи масиву можна ініціалізувати як звичайні змінні. Також можна використовувати список ініціалізації для всього масиву, як це було показано вище. Проте для введених раніше масивів x та y, операція `x = y;` є неприпустимою, навіть при тому, що типи та кількість елементів у цих масивах співпадають. Замість некоректного оператора `x = y;` має бути написаний код, подібний наступному:

```
for( int i = 0; i < 10; i++ ) x[i] = y[i];
```

Одними з найбільш поширених операцій, які часто виконуються над масивами є вставлення або видалення елементів у масиві та сортування його елементів. Розглянемо ці операції більш детально.

Вставлення елементів у масив. У цій лабораторній роботі розглядаються *статичні* масиви, розмір яких визначений *ще на етапі написання програми* і *не може змінюватись* в процесі її виконання (на відміну від динамічних масивів, які будуть розглядатись пізніше). Вставлення елементів в такі масиви, з одночасним збереженням існуючих у них даних, можливе тільки, якщо в масивах є достатньо місця для включення нового елементу. Наприклад, якщо в масив, де вже використовується 10 елементів, необхідно додати ще один елемент, ця операція буде можливою тільки, якщо розмір масиву дорівнює 11 або більше.

Вставлення елементів у масив може відбуватись ліворуч або праворуч від деякого довільного елементу. Приклад реалізації таких операцій наведено у лістингу 4.2.

Лістинг 4.2:

```
 1: #include <stdio.h>
 2:
 3: // Початковий вміст масиву:
 4: // [10, 20, 30, 40, 60, 0, 0]
 5: int ar[7] = { 10, 20, 30, 40, 60 };
 6:
 7: int main( void )
 8: {
 9:     // Індекс елементу, до/після якого слід
```

[1] Ці обмеження можуть бути різними для різних компіляторів. До того ж, деякі з них певною мірою можна обійти у C++ за допомогою технології перевантаження операторів.

```
10:      // вставити новий елемент.
11:      int Index;
12:      // Значення елементу, який слід вставити.
13:      int Value;
14:
15:  // Вивести вміст масиву.
16:      for( int i = 0; i < sizeof(ar)/sizeof(ar[0]);
17:      i++ ) printf("[%d] = %d\n", i, ar[i]);
18:
19:  // Вставити елемент ліворуч від елементу 60.
20:      Index = 4;
21:      Value = 50;
22:      for( int i = sizeof(ar)/sizeof(ar[0]) - 1;
23:      i > Index; i-- )
24:      {
25:          ar[i] = ar[i-1];
26:      }
27:      ar[Index] = Value;
28:
29:  // Вивести вміст масиву.
30:      printf("\n");
31:      for( int i = 0; i < sizeof(ar)/sizeof(ar[0]);
32:      i++ ) printf("[%d] = %d\n", i, ar[i]);
33:
34:  // Вставити елемент праворуч від елементу 60.
35:      Index = 5;
36:      Value = 70;
37:      Index++; // За рахунок "праворуч" індекс +1.
38:      for( int i = sizeof(ar)/sizeof(ar[0]) - 1;
39:      i > Index; i-- )
40:      {
41:          ar[i] = ar[i-1];
42:      }
43:      ar[Index] = Value;
44:
45:  // Вивести вміст масиву.
46:      printf("\n");
47:      for( int i = 0; i < sizeof(ar)/sizeof(ar[0]);
48:      i++ ) printf("[%d] = %d\n", i, ar[i]);
49:
50:      return 0;
51:  }
```

Для вставлення нового елементу у масив обов’язково слід знати

значення цього елементу (задається за допомогою змінної `int Value` у лістингу 4.2), а також індекс елементу в масиві `int Index`, до або після якого вставляється новий елемент.

При вставленні нового елементу ліворуч від вказаного, необхідно зсунути всі елементи, починаючи з вказаного (з індексом `Index`), на одну позицію праворуч. Цю операцію виконує код у рядках 22-26. Після цього, у те місце, яке раніше займав елемент з індексом `Index`, слід записати новий елемент зі значенням `Value`: `ar[Index] = Value;`.

Аналогічним чином працює код для вставлення в масив нового елемента праворуч від елемента із заданим індексом (рядки 37-43). В цьому випадку зсувати праворуч необхідно всі елементи масиву починаючи з елемента, наступного за вказаним. Тобто починаючи з елемента з індексом `Index+1`. Щоб врахувати це, у програмі з лістингу 4.2 виконується операція інкременту індексу `Index++;` (рядок 37). Коли всі потрібні елементи зсунуті праворуч (код рядків 38-42), відбувається ініціалізація нового (вставленого) елементу у масиві: `Value`: `ar[Index] = Value;`.

Код у рядках 16-17, 30-32, 46-48 дозволяє вивести на консоль вміст масиву до та після виконання операцій вставлення елементів:

До вставлення елементів	Після вставлення елементу 50 ліворуч від елементу зі значенням 60	Після вставлення елементу 70 праворуч від елементу зі значенням 60
[0] = 10	[0] = 10	[0] = 10
[1] = 20	[1] = 20	[1] = 20
[2] = 30	[2] = 30	[2] = 30
[3] = 40	[3] = 40	[3] = 40
[4] = 60	[4] = 50	[4] = 50
[5] = 0	[5] = 60	[5] = 60
[6] = 0	[6] = 0	[6] = 70

Видалення елементів з масиву. Код для виконання цієї операції подібний до раніше розглянутого коду вставлення елемента у масив. Для видалення елементу з масиву необхідно зсунути всі елементи в масиві, починаючи з вказаного, на одну позицію ліворуч. Після цього, за необхідності виконати ініціалізацію останнього елементу масиву (наприклад, значенням нуль).

Вставлення/видалення кількох елементів у масиві. Потреба у таких операціях, коли необхідно вставити в масив не один елемент, а групу з декількох елементів, або видалити з масиву групу з кількох елементів, виникає досить часто. Її можна інтерпретувати як вставлення в середину певного масиву іншого масиву. Або як видалення з масиву елементів іншого масиву. Цю операцію, звичайно ж, можна реалізувати шляхом послідовного використання раніше наведеного коду для вставлення/видалення окремих елементів. Однак такий підхід, хоча і є надійним, не завжди є достатньо ефективним. Враховуючи

це наведемо приклад альтернативного коду, який дозволяє вставити елементи одного масиву в інший масив:

Лістинг 4.3:

```
 1: #include <stdio.h>
 2:
 3: int ar[7]  = { 10, 20, 60, 70 };
 4: int b [3]  = { 30, 40, 50 };
 5:
 6: int arSize = sizeof(ar)/sizeof(ar[0]);
 7: int bSize  = sizeof(b)/sizeof(b[0]);
 8:
 9: int main( void )
10: {
11:     int Index, i, n;
12:
13: // Вивести вміст масиву ar.
14:     for( i = 0; i < arSize; i++ )
15:         printf("[%d] = %d\n", i, ar[i]);
16:
17: // Вставити масив b після елементу 20
18: // в масиві ar.
19:     Index = 2;
20:     // Зсунути елементи в масиві ar.
21:     for( i = Index; i < arSize; i++ )
22:     {
23:         n = i + bSize;
24:         if( n < arSize ) ar[n] = ar[i];
25:         else break;
26:     }
27:     // Вставити масив b.
28:     for( i = 0; i < bSize; i++ )
29:     {
30:         n = Index + i;
31:         if( n < arSize ) ar[n] = b[i];
32:         else break;
33:     }
34:
35: // Вивести вміст масиву.
36:     printf("\n");
37:     for( i = 0; i < arSize; i++ )
38:         printf("[%d] = %d\n", i, ar[i]);
39:
40:     return 0;
```

```
41: }
```

У рядках 3-4 визначаються масиви `ar` та `b`, розміри яких розраховуються і зберігаються у глобальних змінних `arSize` та `bSize` у рядках 6-7.

Спочатку в програмі виводиться вміст масиву `ar` до виконання операції вставлення елементів (рядки 14-15). При цьому на консоль виводиться наступна інформація:

```
[0] = 10
[1] = 20
[2] = 60
[3] = 70
[4] = 0
[5] = 0
[6] = 0
```

Вставлення масиву `b` всередину масиву `ar` виконується в два етапи. Спочатку всі елементи масиву `ar`, починаючи з третього (з індексом 2) зсуваються праворуч на кількість позицій, що дорівнює кількості елементів у масиві `b` (рядки 21-26). Потім «вільні місця», які утворились при зсуві елементів масиву `ar`, заповнюються елементами масиву `b` (рядки 28-33). Після цього на консоль виводиться вміст зміненого масиву `ar`:

```
[0] = 10
[1] = 20
[2] = 30
[3] = 40
[4] = 50
[5] = 60
[6] = 70
```

Аналогічним чином реалізується видалення кількох елементів з масиву.

Сортування елементів у масиві. ***Сортування даних*** – це операція впорядкування даних за певним принципом (за деякою ознакою, в певному порядку). Сортування даних є однією з найбільш часто вживаних операцій, які застосовуються до масивів даних. На даний момент існує велика кількість алгоритмів сортування даних. Всі ці алгоритми *не змінюють самі дані*, проте можуть змінювати порядок розташування цих даних. Далі розглянемо лише два найпростіших випадки: алгоритми бульбашкового та вибіркового сортування.

Алгоритм *бульбашкового сортування* (bubble sort) зводиться до виконання наступних кроків. По черзі перебираємо всі елементи масиву. Якщо два

сусідніх елементи масиву не задовольняють вказаному критерію сортування, вони міняються місцями. В результаті вже після першого проходу всього масиву, його крайній елемент (лівий або правий, залежно від умов сортування) займає потрібне місце. Після цього алгоритм продовжує працювати з усіма елементами окрім згаданого крайнього. Вже після другого проходу такого підмасиву елементів (без врахування крайнього елементу), правильне положення поблизу крайнього елементу займає другий елемент. Потім алгоритм продовжує роботу з усіма елементами, окрім вказаних двох, і так далі. Приклад реалізації цього алгоритму наведений у лістингу 4.4.

Лістинг 4.4:

```
 1: #include <stdio.h>
 2:
 3: int ar[7]  = { -1, 5, -10, 6, 3, 2, 41 };
 4:
 5: int arSize = sizeof(ar)/sizeof(ar[0]);
 6:
 7: int main( void )
 8: {
 9:     int i, j, tmp;
10:
11: // Вивести вміст масиву ar.
12:     for( i = 0; i < arSize; i++ )
13:         printf("%d ", ar[i]);
14:
15: // Бульбашкове сортування.
16:     // Проходимо масив в прямому напрямку,
17:     // проглядаючи всі елементи окрім останнього.
18:     for( i = 0; i < arSize-1; ++i )
19:     {
20:         // Проходимо підмасив у зворотному
21:         // напрямку, починаючи з поточного
22:         // елементу.
23:         for( j = arSize-1; j > i; --j )
24:         {
25:             // Порівнюємо два сусідніх елементи.
26:             if( ar[j-1] > ar[j] )
27:             {
28:             // Якщо потрібно = переставляємо їх.
29:                 tmp = ar[j-1];
30:                 ar[j-1] = ar[j];
31:                 ar[j] = tmp;
32:             }
33:         }
```

```
34:      }
35:
36:  // Вивести вміст масиву.
37:      printf("\n");
38:      for( i = 0; i < arSize; i++ )
39:          printf("%d ", ar[i]);
40:      printf("\n");
41:
42:      return 0;
43:  }
```

Оскільки в основі алгоритму бульбашкового сортування лежить операція порівняння двох сусідніх елементів, щоб виконати сортування даних достатньо пройтись по всьому масиву елементів *за виключенням останнього елементу* (див. рядок 18). Для масиву `ar` з лістингу 4.4 останній елемент масиву має індекс `arSize-1`. При проходженні по масиву цей останній елемент однак *не буде «забутим»*, оскільки його значення буде порівнюватись із попереднім елементом з індексом `arSize-2` при виконанні зовнішнього циклу `for( i = 0; i < arSize-1; ++i )`.

Для деякого поточного елементу з індексом `i` у масиві відбувається попарне порівняння всіх елементів у підмасиві, що починається з елемента, наступного після елемента з індексом `i` (цикл `for` у рядках 23-33). Якщо відповідні пари елементів не задовольняють критерій сортування (рядок 26), вони міняються місцями (рядки 29-31).

Код, наведений у лістингу 4.4, виконує сортування елементів в порядку зростання їх величини. Ось результати, які виводяться на консоль до сортування масиву `ar` і після виконання операції сортування (код у рядках 12-13, 37-40):

```
До сортування:    –1 5 –10 6 3 2 41
Після сортування: –10 –1 2 3 5 6 41
```

Алгоритм *вибіркового сортування* (selection sort), можливо, навіть більш простий, ніж алгоритм бульбашкового сортування. Принцип його дії наступний. Алгоритм проходить весь масив елементів і знаходить той елемент, який найкраще задовольняє заданому критерію. Цей елемент відразу ж міняється місцями з першим (або останнім) елементом масиву. Потім алгоритм повторює цей процес для всіх елементів, окрім першого (останнього). В результаті під час другої ітерації на потрібному місці опиняється вже другий елемент масиву. Повторюючи тепер цей алгоритм ще раз і ще раз, поступово розташовуємо елементи згідно вказаного критерія. Приклад реалізації цього алгоритму наведений у лістингу 4.5.

Лістинг 4.5:

```
 1: #include <stdio.h>
 2:
 3: int ar[7]  = { -1, 5, -10, 6, 3, 2, 41 };
 4:
 5: int arSize = sizeof(ar)/sizeof(ar[0]);
 6:
 7: int main( void )
 8: {
 9:     int i, j, minIndex, minElem, tmp;
10:
11: // Вивести вміст масиву ar.
12:     for( i = 0; i < arSize; i++ )
13:         printf("%d ", ar[i]);
14:
15: // Вибіркове сортування.
16:     // Проходимо масив в прямому напрямку.
17:     for( i = 0; i < arSize-1; i++ )
18:     {
19:         minIndex = i;
20:         minElem = ar[i];
21:         // Знаходимо найменший елемент.
22:         for( j = i+1; j < arSize; j++ )
23:         {
24:             tmp = ar[j];
25:             // Порівнюємо елементи.
26:             if( tmp < minElem )
27:             {
28:                 minElem  = tmp;
29:                 minIndex = j;
30:             }
31:         }
32:         // У попередньому циклі for знайдено
33:         // мінімальний елемент.
34:         if( i != minIndex )
35:         {   // Встановити його на потрібне місце.
36:             tmp           = ar[i];
37:             ar[i]         = minElem;
38:             ar[minIndex] = tmp;
39:         }
40:     }
41:
42: // Вивести вміст масиву.
43:     printf("\n");
```

```
44:       for( i = 0; i < arSize; i++ )
45:           printf("%d ", ar[i]);
46:       printf("\n");
47:
48:       return 0;
49:   }
```

У циклі `for` (див. рядки 17-40 у лістингу 4.5) проходимо масив `ar` в прямому напрямку, проглядаючи всі елементи масиву, починаючи з першого елементу (з індексом `0`) і закінчуючи передостаннім (з індексом `arSize-2`). Індекс `i` в цьому циклі є як номером елементу у початковому масиві, так і індексом елемента, який має зайняти правильне місце у відсортованому масиві. Для кожного елемента з індексом `i` розглядається підмасив всіх елементів, починаючи з поточного і до кінця масиву `ar` (рядки 22-31). У цьому підмасиві знаходиться елемент з мінімальним значенням, який переміщується на початок підмасиву – в положення, що визначається індексом `i` (рядки 34-39). Після цього код переходить до наступної ітерації для елементу з індексом `i+1`.

Результат роботи програми з лістингу 4.5 такий же як і для програми з лістингу 4.4 – масив `ar` виявляється відсортованим в порядку збільшення значень елементів: –10 –1 2 3 5 6 41.

Багатовимірні масиви

Масиви можуть бути ***багатовимірними***. Такі багатовимірні масиви можна розглядати як масиви масивів. Синтаксис оголошення багатовимірного масиву наступний:

```
Тип_даних Ім'я_масиву[Розмір_1][Розмір_2]…[Розмір_N];
```

Константи `Розмір_1`, `Розмір_2`, … `Розмір_N` визначають розміри багатовимірного масиву. Ось приклади оголошення багатовимірних масивів:

```
int a[2][3];
int b[3][2];
int c[2][3][4];
```

Масиви `a` та `b` містять однакову кількість (6) елементів. Масив `a` являє собою пару масивів, кожен з яких містить по 3 елементи типу `int`. Масив `b` містить три масиви, кожен з яких зберігає пару значень типу `int`. Таким чином, масиви `a` та `b` можна ототожнити з матрицями цілих чисел з розмірностями 2×3 та 3×2. Масив `c` являє собою масив з двох матриць

розмірністю 3×4 (всього 24 елементи).

Всі елементи багатовимірного масиву розташовуються в пам'яті послідовно, так само, як і елементи одновимірних масивів. При цьому кожен елемент багатовимірного масиву характеризується унікальним набором індексів, кількість яких відповідає розмірності масиву (кількості його розмірів). Розташування елементів масивів у пам'яті відбувається таким чином, що найшвидше змінюється останній (найбільш правий) індекс, що відповідає останньому розміру масиву. Наприклад, для того щоб звернутись до елемента з індексами `i` та `j` в масивах `a` та `b` використовується конструкція `a[i][j]`, `b[i][j]`. Для тривимірного масиву `c` фрагмент `c[i][j][k]` позначає елемент з індексом `i`, що відповідає першому розміру масиву, індексами `j` та `k`, що відповідають другому та третьому розмірам масиву. При цьому при послідовному переборі всіх елементів масиву, найбільш швидко змінюється зовнішній (більш правий) індекс – для масиву `c` це індекс `k`; потім починає змінюватись індекс `j`, і найбільш повільно змінюється індекс `i`.

Подібно одновимірним масивам, вміст багатовимірних масивів можна задавати за допомогою списку ініціалізації. При цьому використовуються ті ж самі правила як і для одновимірних масивів але з урахуванням правил зміни індексів. Ось приклад ініціалізації раніше наведених масивів `a`, `b` та `c`:

```
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
int b[3][2] = { {1, 2}, {3, 4}, {5, 6} };
int c[2][3][4] =
{
     {
          {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}
     },
     {
          {12, 11, 10, 9}, {8, 7, 6, 5}, {4, 3, 2, 1}
     }
};
```

Як видно, перший розмір масиву визначає кількість блоків в списку ініціалізації, що обмежений дужками `{}`. Для масивів `a` та `c` це 2 блоки, для масиву `b` – 3 блоки. В кожному з цих блоків наводяться або значення елементів (для масивів `a` та `b`) або додаткові блоки (для масиву `c`). Кількість елементів у блоках відповідає другому розміру масиву – так дані для масиву `a` задані групами по три числа типу `int`, що відповідає другому розміру 3 масиву `a[2][3]`. Для масиву `b[3][2]` це групи по 2 числа типу `int`. Найбільш складно виглядає список ініціалізації для масиву `c[2][3][4]` – це список з двома блоками, кожен з яких містить три підблоки, в кожному по 4 числа типу `int`.

Виходячи із даних, заданих для масивів `a`, `b` та `c`, легко переконатись,

що елемент `a[1][2]` містить число 2, елемент `b[2][2]` містить число 4, а елемент `c[1][2][3]` містить число 7.

Робота з багатовимірними масивами відбувається за тими ж правилами, що й для одновимірних масивів, з урахуванням правил зміни індексів, які розглядались вище. Всі операції над елементами багатовимірного масиву реалізуються ідентично до операцій над елементами того ж типу в одновимірному масиві. У лістингу 4.6 наводиться приклад роботи з двовимірними масивами.

Лістинг 4.6:

```
 1: #include <stdio.h>
 2:
 3: int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
 4: int b[3][2] = { 0 };
 5:
 6: int main( void )
 7: {
 8:     int i, j;
 9:
10: // Вивести вміст масиву b.
11:     printf("\t[0]\t[1]\n");
12:     for( i = 0; i < 3; i++ )
13:     {
14:         printf("[%d]: ", i);
15:         for( j = 0; j < 2; j++ )
16:         {
17:             printf("\t%d", b[i][j]);
18:         }
19:         printf("\n");
20:     }
21:
22: // Транспонування матриці a
23: // і запис результату в b.
24:     for( i = 0; i < 2; i++ )
25:     {
26:         for( j = 0; j < 3; j++ )
27:         {
28:             b[j][i] = a[i][j];
29:         }
30:     }
31:
32: // Вивести вміст масиву.
33:     printf("\n");
34:     // Вивести вміст масиву b.
35:     printf("\t[0]\t[1]\n");
```

```
36:       for( i = 0; i < 3; i++ )
37:       {
38:            printf("[%d]: ", i);
39:            for( j = 0; j < 2; j++ )
40:            {
41:                 printf("\t%d", b[i][j]);
42:            }
43:            printf("\n");
44:       }
45:       printf("\n\n");
46:
47:       return 0;
48:  }
```

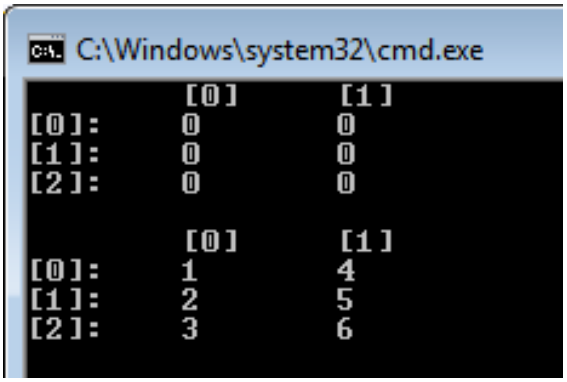


Рис. 21

У рядку 3 ініціалізується масив `a[2][3]`, а в рядку 4 масив `b[3][2]` заповнюється нулями. У рядках 24-30 відбувається транспонування матриці, пов'язаної з масивом `a[2][3]`, причому отримана транспонована матриця записується в масив `b[3][2]`. Код у рядках 11-20 та 33-45 використовується для виведення результату роботи програми – вмісту масиву `b` – на екран (рис. 21). Перший блок даних показує вміст `b[3][2]` до виконання транспонування, другий – після виконання цієї операції.

Текстові (символьні) рядки

Текстовим (символьним) рядком називають довільну послідовність символів. У мові C/C++ часто використовуються т. зв. ***текстові рядки в форматі C***. Вони являють собою послідовність символів, що закінчується спеціальним ***нульовим символом***, який відмічає кінець рядка.

У мові програмування C/C++ немає вбудованого типу для текстового рядка[1]. Такі рядки типово розглядаються як масиви символів типу `char` або широких символів типу `wchar_t`. Відповідно, з таких символів будуються звичайні текстові рядки (з символами типу `char`) та широкі або UNICODE-рядки

[1] У стандартній бібліотеці C++ є тип даних `string` для роботи з текстовими рядками, проте він не є вбудованим. Він стає доступним при підключенні файлу `<string>` або `<string.h>`.

(на основі символів типу `wchar_t`). Для звичайних рядків нульовий символ є символом `'\0'`, а для широких рядків цей символ визначається як `L'\0'`. У будь-якому представленні код нульового символу дорівнює нулю.

Робота з текстовими рядками у C/C++ майже нічим не відрізняється від роботи з масивами. При роботі з рядками необхідно пам'ятати лише дві їх особливості:

1) в кінці текстового рядка в форматі C обов'язково має бути присутній нульовий символ;

2) стандартні функції бібліотек C/C++, які працюють з рядками в форматі C, як правило, ігнорують символи, які знаходяться після нульового символу.

Для збереження текстового рядка, що містить N значущих символів, необхідно створити масив з розміром як мінімум N+1 символів, оскільки додатковий символ (+1) буде потрібний для збереження нульового символу.

Ініціалізація текстових рядків може відбуватись як за загальними правилами (як для символьних масивів, шляхом вказування окремих символів), так і за допомогою текстових констант. Нижче наведено можливі приклади визначення та ініціалізації текстових рядків:

```
char cText1[8] = {'U','k','r','a','i','n','e','\0'};
char cText2[8] = "Ukraine";
wchar_t wcText3[8] = L"Ukraine";
wchar_t wcText4[8] = { L'U', L'A', L'\0'};
```

Текстовий рядок `cText1` задається посимвольно, а текстовий рядок `cText2` ініціалізується текстовою константою `"Ukraine"`. Аналогічно виконується ініціалізація широких текстових рядків `wcText3` та `wcText4`, причому в останньому випадку в рядку використовуються лише перші 2 символи, а інші шість – ігноруються (вони дорівнюють нулю).

З текстовими рядками можна працювати як з масивами. Приклад програми, що виконує різні операції над рядками як над масивами символів, наведено у лістингу 4.7.

Лістинг 4.7:

```
1:  #include <stdio.h>
2:
3:  char s[8] = "Ukraine";
4:
5:  int main( void )
6:  {
7:      int i;
8:
9:      // Вивести вміст рядка.
```

```
10:      printf("%s\n", s);
11:
12:      // Перетворити рядок.
13:      for( i = 0; i < 7; i++ )
14:          if( i % 2 ) s[i] = s[i] + ('A'-'a');
15:
16:      // Вивести вміст рядка.
17:      printf("\n%s\n", s);
18:
19:      s[2] = 'R';
20:      s[3] = '\0';
21:      printf("\n%s\n\n", s);
22:
23:      return 0;
24:  }
```

У рядку 3 задається та ініціалізується текстовий рядок `s`. У рядках 13-14 кожен символ з непарним індексом (1, 3 та 5) змінюються на величину `'A'-'a'`, що відповідає переходу від малих букв до великих. В результаті початкове слово `"Ukraine"`, записане у рядку, змінюється на `"UKrAiNe"`. Потім буква `'r'` замінюється на `'R'`, а замість символу `'a'` вставляється нульовий символ `'\0'` (рядки 19, 20). В результаті текстовий рядок буде вже сприйматись як `"UKR"`.

Для роботи з текстовими рядками в форматі С існує велика кількість функцій, які входять до стандартної бібліотеки C/C++. Вони стають доступними при підключенні файлу заголовку `string.h`. Наведемо, стислий опис деяких найбільш популярних функцій:

`strlen(s)` – повертає довжину рядка `s` – кількість символів у рядку (без урахування нульового символу).

`strcmp(s1, s2)` – виконує порівняння рядків `s1` та `s2`. Повертає від'ємне значення, якщо `s1 < s2`; нуль – якщо `s1` дорівнює `s2`; додатне значення, якщо `s1 > s2`.

`strcpy(s, s1)` – копіює рядок `s1` у рядок `s`. Ця функція є аналогом оператора присвоювання для рядків.

`strncpy(s, s1, n)` – копіює `n` символів з рядка `s1` у рядок `s`.

`strcat(s, s1)` – приєднує рядок `s1` до рядка `s`.

Більш детальну інформацію про роботу з текстовими рядками можна знайти в онлайн-документації мов C/C++.

ЗАВДАННЯ

Варіант 1

1) Задати масив цілих чисел `M`, що складається з 5 елементів Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` так, щоб кожен елемент виводився з нового рядка.
3) Реалізувати перевантажені функції, які розраховують квадрат цілого та дійсного числа і відповідно повертають результат як ціле або дійсне число. Використовуючи ці функції, реалізувати ще одну функцію, яка б для заданого масиву розраховувала дисперсію елементів за формулами:
$$\bar{M} = \frac{M[0] + M[1] + ...M[4]}{5},\ D^2 = \frac{(M[0] - \bar{M})^2 + (M[1] - \bar{M})^2 + ...(M[4] - \bar{M})^2}{5}.$$
4) Вивести на екран величину `D`.
5) Відсортувати елементи масиву `M` за зростанням та вивести результат на екран.
6) Створити функцію, яка вставляє до початкового масиву ще один елемент на початку масиву, та вивести результат її роботи на екран.

Варіант 2

1) Задати масив цілих чисел `M`, що складається з 5 елементів Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи масиву комами та пробілом.
3) Створити функцію, яка обчислює логарифм числа x по довільній основі A, і має два аргументи: дійсне число x та ціле A. При цьому за замовчуванням логарифм має обчислюватись по основі 2.
4) Використовуючи функцію з пункту 3, розрахувати логарифм від кожного елемента масиву, створивши новий масив `M1` за правилом: `M1[n]= log(n+2)M[n]`. Вивести вміст масиву `M1` на консоль.
5) Визначити максимальний та мінімальний із елементів масиву `M1` та вивести їх значення на консоль.
6) Відсортувати елементи масиву `M1` за спаданням та вивести результат на консоль.
7) Створити функцію, яка вставляє до початкового масиву `M` ще один елемент в кінець масиву, та вивести вміст масиву `M` на консоль.

Варіант 3

1) Задати масив цілих чисел `M1`, що складається з 5 елементів. Записати в `M1` елементи масиву, які вводяться з клавіатури. Аналогічним чином задати масив `M2`.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M1` в один рядок, розділяючи елементи знаком табуляції.
3) Записати перевантажену функцію `Sum()`, яка обчислює суму двох чисел, якщо вхідні параметри мають цілий тип, та поелементну суму двох масивів, якщо вхідні параметри – це масиви.
4) Розрахувати і вивести на екран результат роботи `Sum(M1,M2)`.
5) Відсортувати елементи масиву `M1`, `M2` за зростанням та вивести результат на екран.
6) Створити функцію, яка вставляє до одного із початкових масивів ще один елемент на початку масиву, та вивести результат її роботи на екран.

Варіант 4

1) Задати масив цілих чисел `M`, що складається з 5 елементів Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M`, так, щоб кожен елемент виводився у новому рядку з додатковим відступом – пробілами, кількість яких дорівнює індексу елемента.
3) Створити дві функції, які реалізують вставку довільного числа на початку або в кінці масиву.
4) Отримати масив, в який вставлені нові елементи на початку та в кінці масиву. Вивести його вміст на екран.
5) Відсортувати елементи отриманого масиву за спаданням та вивести результат на екран.
6) Створити функцію, яка вставляє до початкового масиву ще один елемент в довільне місце масиву, та вивести результат її роботи на екран.

Варіант 5

1) Задати символьний рядок (масив символів) `M`, що складається з 5 елементів. Елементи масиву мають вводитись з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, не розділяючи елементи.
3) Створити функцію, яка виконує інверсію заданого рядка (переписує його задом наперед). Вивести результат виконання цього перетворення на екран.
4) Відсортувати елементи отриманого масиву за абеткою та вивести результат на екран. Скористайтесь особливостями розташування символів

абетки в таблиці символів.

5) Створити функцію, яка вставляє до початкового масиву ще один елемент на його початку, і вивести отриманий результат на екран.

Варіант 6

1) Задати масив цілих чисел `M`, що складається з 6 елементів Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи крапкою з комою та пробілом.
3) Створити функції, які визначають мінімальне та максимальне значення окремо серед компонент масиву з парними або непарними індексами. Вивести на екран результат їх роботи з відповідними коментарями.
4) Відсортувати елементи масиву `M` за зростанням та вивести на екран вміст відсортованого масиву.
5) Створити функцію, яка вставляє до початкового масиву ще один елемент на його початку; вивести результат роботи цієї функції на екран.

Варіант 7

1) Задати символьний рядок (масив символів) `M1`, що складається з 5 елементів. Елементи масиву мають вводитись з клавіатури. Аналогічним чином задати рядок `M2`.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M1`, в один рядок, розділяючи елементи символом `'*'`.
3) Перевантажити функцію `Sum()` таким чином, щоб вона обчислювала суму двох чисел, якщо вхідні параметри відносяться до цілого типу, та об'єднувала два символи в текстовий рядок, якщо вхідні параметри – це символи.
4) Застосувати функцію з пункту 3 до масивів `M1` та `M2` поелементно і вивести результат на екран.
5) Виконати завдання пункту 4, використовуючи приведення типу елементів символьних масивів до типу `int`.
6) Створити функцію, яка вставляє до початкового масиву ще один елемент в кінець масиву; вивести результат її роботи на екран.

Варіант 8

1) Задати масив цілих чисел `M`, що складається з 6 елементів Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи знаком табуляції.

3) Створити функцію, яка реалізовує операцію піднесення до цілого степеня.
4) Використовуючи функцію із попереднього пункту, піднести всі парні елементи масиву до квадрату, а всі непарні – до кубу. Вивести на екран результат розрахунку.
5) Знайти максимальний та мінімальний елементи отриманого масиву та вивести їх значення.
6) Відсортувати елементи масиву `M` за зменшенням та вивести вміст відсортованого масиву на екран.
7) Створити функцію, яка вставляє до масиву `M` ще один елемент посередині масиву, та вивести результат її роботи на екран.

Варіант 9

1) Задати змінну `M` для масиву цілих чисел, що складається з 100 елементів. Записати у масив `M` елементи згідно формули `M[n] = cos(2.0*3.1415926535*n/100.0)`.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` так, щоб кожен елемент виводився у новому рядку із зазначенням індексу елемента.
3) Знайти максимальні значення серед парних та серед непарних елементів отриманого масиву, а також їх індекси, та вивести цю інформацію на екран.
4) Відсортувати елементи масиву `M` за зростанням. Вивести результат на екран.

Варіант 10

1) Задати символьний рядок (масив символів) `M`, що складається з 6 елементів. Елементи масиву мають вводитись з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, не розділяючи елементи.
3) Написати програму для перевірки наявності заданого символу у масиві `M` (символ вводиться з клавіатури). Вивести на екран результат такої перевірки.
4) Відсортувати елементи масиву `M` за абеткою у зворотному порядку та вивести отриманий результат на екран. Скористайтесь особливостями розташування символів абетки в таблиці символів.
5) Створити функцію, яка вставляє до початкового масиву ще один елемент в кінець масиву. Вивести вміст масиву на екран.

Варіант 11

1) Задати масив цілих чисел M1, що складається з 5 елементів, які вводяться з клавіатури. Аналогічним чином задати масив M2.
2) Реалізувати функцію, яка виводить на екран елементи масиву M1 в один рядок, розділяючи елементи двокрапкою та пробілом.
3) Реалізувати перевантажену функцію Product(), яка рахує добуток двох чисел, якщо вхідні параметри мають дійсний тип, та суму добутків елементів з однаковими індексами двох масивів $\sum_i M1[i] \cdot M2[i]$, якщо вхідні параметри – це масиви цілих чисел.
4) Розрахувати і вивести на екран результат роботи Product(M1,M2).
5) Відсортувати елементи масиву M1 за зростанням та вивести отриманий результат на екран.
6) Відсортувати елементи масиву M2 за спаданням та вивести отриманий результат на екран.

Варіант 12

1) Задати масив цілих чисел M1, що складається з 5 елементів, які вводяться з клавіатури. Аналогічним чином задати масив дійсних чисел M2.
2) Реалізувати функцію, яка виводить на екран елементи масиву в один рядок, розділяючи елементи знаком тире «–». Вивести вміст масивів M1 та M2.
3) Записати перевантажену функцію Sum(), яка рахує суму двох чисел, і повертає результат цілого типу, якщо вхідні параметри є цілими числами, та повертає аналогічний результат дійсного типу, якщо хоча б один із вхідних параметрів є змінною дійсного типу.
4) Використовуючи перевантажену функцію з пункту 3, розрахувати та вивести на екран поелементну суму масивів M1 та M1, M1 та M2.
5) Відсортувати елементи масиву M1 за зростанням та вивести отриманий результат на екран.
6) Створити функцію, яка вставляє до масиву M2 ще один елемент в кінці масиву, та вивести результат її роботи на екран.

Варіант 13

1) Задати масив цілих чисел M1, що складається з 5 елементів, які вводяться з клавіатури. Аналогічним чином задати масив M2.
2) Реалізувати функцію, яка виводить на екран елементи масиву в один рядок, розділяючи елементи знаком «/» із пробілами ліворуч та праворуч від цього знаку. Вивести вміст масивів M1 та M2.
3) Отримати новий масив M3, парні елементи якого є сумою відповідних

елементів масивів `M1` та `M2`, а непарні – різницею.

4) Вивести на екран вміст масиву `M3`.
5) Визначити найбільший та найменший елементи масиву `M3` та їх індекси і вивести ці дані на екран з відповідними коментарями.
6) Відсортувати елементи масиву `M3` за зростанням та вивести отриманий результат на екран.

Варіант 14

1) Задати масив цілих чисел `M1`, що складається з 6 елементів, які вводяться з клавіатури. Аналогічним чином задати масив дійсних чисел `M2`.
2) Реалізувати функцію, яка виводить на екран елементи масиву так, щоб кожен елемент виводився з нового рядка з вказанням індексу. Вивести одночасно вміст масивів `M1` та `M2`.
3) Створити перевантажену функцію `Divide()`, яка містить два аргументи, і повертає результат ділення першого аргументу на другий, причому результат належить до цілого типу, якщо перший аргумент є цілим числом, та до дійсного типу, якщо перший аргумент є змінною дійсного типу.
4) Використовуючи перевантажену функцію з пункту 3, розрахувати та вивести на екран результат ділення елементів масивів `M1` та `M2` на їх порядковий номер у масиві (який починається з 1, не індекс!).
5) Створити функцію, яка вставляє до масивів `M1` та `M2` ще один елемент посередині масиву, та вивести результат її роботи на екран.

Варіант 15

1) Задати символьний рядок (масив символів) `M`, що складається з 6 елементів. Елементи масиву мають вводитись з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи знаком дефісу.
3) Створити функцію, яка заміняє кожен символ масиву на наступний символ абетки (наприклад, `'a'` на `'b'`, `'b'` на `'c'`, `'c'` на `'d'`, `'в'` на `'г'` тощо). Скористайтесь особливостями розташування символів абетки в таблиці символів. Вивести результат такого перетворення на екран.
4) Визначити, чи є в масиві `M` однакові елементи та вивести результат на екран з відповідними коментарями.
5) Створити функцію, яка вставляє до початкового масиву ще один елемент посередині масиву, та вивести результат її роботи на екран.

Варіант 16

1) Задати масив цілих чисел `M`, що складається з 5 елементів. Записати в `M` елементи масиву, які вводяться з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи текстовим фрагментом `" * "`.
3) Створити власну функцію, яка реалізовує операцію піднесення числа до цілого степеня.
4) Використовуючи функцію із попереднього пункту, піднести всі елементи масиву до степеня, який дорівнює індексу елемента. Вивести на екран результат розрахунку.
5) Знайти максимальний та мінімальний елементи масиву та вивести їх значення на екран.
6) Відсортувати елементи масиву `M` за спаданням та вивести отриманий результат на екран.
7) Створити функцію, яка вставляє до масиву `M` ще один елемент посередині масиву, та вивести результат її роботи на екран.

Варіант 17

1) Задати масив цілих чисел `M1`, що складається з 7 елементів, які вводяться з клавіатури. Аналогічним чином задати масив дійсних чисел `M2`.
2) Реалізувати функцію, яка виводить на екран елементи масиву в один рядок, розділяючи елементи знаком пробілу. Вивести вміст масивів `M1` та `M2`.
3) Записати перевантажену функцію `Mul()`, яка обчислює добуток двох чисел, і повертає результат цілого типу, якщо вхідні параметри є цілими числами, та повертає результат дійсного типу, якщо хоча б один із вхідних параметрів є змінною дійсного типу.
4) Використовуючи перевантажену функцію з пункту 3, розрахувати та вивести на екран поелементний добуток масивів `M1` та `M2`.
5) Відсортувати елементи масиву `M1` за спаданням та вивести отриманий результат на екран.
6) Створити функцію, яка вставляє до масиву `M2` ще один елемент в кінці масиву, та вивести результат її роботи на екран.

Варіант 18

1) Задати масиви дійсних чисел `M1`, `M2`, що складаються з 10 елементів: `M1[n]=1.0/(1.0+n)`, `M2[n]=2.0/(2.0+n)`.
2) Реалізувати функцію, яка виводить на екран елементи масиву в один рядок, розділяючи елементи фрагментом `" : "`. Вивести зміст масивів `M1` та `M2`.

3) Записати перевантажену функцію `Sum()`, яка обчислює суму двох чисел, і повертає результат цілого типу, якщо вхідні параметри є цілими числами, та повертає результат дійсного типу, якщо хоча б один із вхідних параметрів є змінною дійсного типу.
4) Відсортувати елементи масиву `M1` за зростанням, а `M2` – за спаданням та вивести отриманий результат на екран. Використовуючи перевантажену функцію з пункту 3, розрахувати та вивести на екран поелементну суму відсортованих масивів `M1` та `M2`.
5) Створити функцію, яка вставляє до масиву `M1` ще один елемент в кінці масиву (значення згідно формули), та вивести результат її роботи на екран.

Варіант 19

1) Задати символьний рядок (масив символів) `M`, що складається з 5 елементів. Елементи масиву мають вводитись з клавіатури.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M` в один рядок, розділяючи елементи знаком пробілу.
3) Створити функцію, яка замінює кожен символ масиву на симетричний з алфавіту (наприклад, `'a'` на `'z'`, `'b'` на `'y'`, `'c'` на `'x'`, `'б'` на `'ю'` тощо). Скористайтесь особливостями розташування символів абетки в таблиці символів. Вивести результат виконаного перетворення на екран.
4) Визначити, чи є в масиві `M` однакові елементи та вивести результат на екран з відповідними коментарями.
5) Створити функцію, яка вставляє до початкового масиву ще один елемент посередині масиву (символ вводиться з клавіатури), та вивести результат її роботи на екран.

Варіант 20

1) Задати масив цілих чисел `M1`, що складається з 5 елементів, які вводяться з клавіатури. Аналогічним чином задати масив `M2`.
2) Реалізувати функцію, яка виводить на екран елементи масиву `M1` в один рядок, розділяючи елементи фрагментом `" | "`.
3) Створити перевантажену функцію `Add()`. Ця функція має записувати число, що складається з послідовності цифр двох чисел, якщо вхідні параметри мають цілий тип (наприклад, 8 та 23 дають «823»). Якщо ж вхідні параметри – це масиви, `Add()` має обчислювати аналогічну величину для елементів масивів з однаковими індексами.
4) Розрахувати і вивести на екран результат роботи `Add(M1,M2)`.
5) Відсортувати елементи масиву, який створюється `Add(M1,M2)`, за спаданням та вивести вміст відсортованого масиву на екран.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке масив? Який синтаксис оголошення масиву в C/C++?
2. Що спільного і у чому різниця між масивом і звичайною змінною того ж типу?
3. Чи можна ініціалізувати елементи масивів у момент оголошення масиву? Якщо відповідь позитивна, наведіть приклади ініціалізації масивів.
4. Чи може бути створений масив масивів? Якщо відповідь позитивна, то за яким принципом розташовуються елементи такого об’єкта в пам’яті?
5. Чим багатовимірні масиви відрізняються від одновимірних?
6. Яким чином можна зчитувати та змінювати значення елементів масиву?
7. Чим відрізняються текстові рядки та одновимірні масиви?
8. Чи може довжина текстового рядка бути меншою за розмір відповідного символьного масиву? Наведіть приклади.
9. Як можна визначити довжину текстового рядка в форматі C?
10. Чи можна вставляти нові елементи до статичних масивів? Якщо відповідь позитивна, поясніть за яких умов це можна робити.
11. Наведіть відомі вам алгоритми сортування даних. В чому їх суть?
12. Які операції можна здійснювати з елементами масивів та самими масивами?
13. Як присвоїти елементам одного масиву значення елементів з іншого масиву того ж типу та розміру?

Лабораторна робота № 5.
Вказівники та посилання.
Робота з динамічною пам'яттю

Мета роботи:

1) Ознайомитись з поняттям вказівника та посилання. Навчитись працювати з вказівниками та посиланнями.
2) Навчитись працювати з масивами за допомогою вказівників.
3) Навчитись створювати та використовувати об'єкти, що розміщуються в динамічній пам'яті.

ВКАЗІВНИКИ

Оголошення, визначення та переініціалізація вказівників

Вказівник (покажчик, англ. pointer) – це змінна, яка може містити *адресу* певного об'єкту (змінної, масиву, функції), що розташовується в пам'яті. Ця ***адреса*** являє собою унікальне число, яке дозволяє однозначно визначити положення певного об'єкту або його складових в адресному просторі програми. У мовах C/C++ визначення та використання адрес дозволяється тільки для об'єктів, що розміщуються в оперативній пам'яті. Визначення та використання адрес для об'єктів, які розміщуються в регістрах процесора, не дозволяється[1].

Адреса об'єкта в пам'яті є аналогом поштової адреси будинка. Знаючи адресу об'єкта можна знайти потрібний об'єкт, так само, як знаючи поштову адресу потрібного будинка, можна знайти цей будинок.

У мові C/C++ синтаксис оголошення/визначення вказівника аналогічний до синтаксису оголошення/визначення змінної:

```
Тип* Ім'я_вказівника = Значення;
```

або

```
Тип *Ім'я_вказівника = Значення;
```

`Тип` задає тип тої змінної, на яку може вказувати вказівник. Те, що це не просто змінна певного `Типу`, а *саме вказівник* на змінну цього `Типу`, позначає додатковий знак зірочки «`*`». Цей знак можна писати як відразу ж після `Типу`

[1] Разом з тим, у мовах C/C++ можна напряму звертатись до регістрів процесора, використовуючи або код *вбудованого асемблера* (inline assembler), або т. зв. *псевдорегістри*.

змінної, так і перед ідентифікатором змінної-вказівника – `Ім'ям_вказівника`.

Незалежно від конкретного `Типу` змінної, розмір змінної-вказівника – тобто розмір специфічного типу `Тип*` – в сучасних системах завжди дорівнює 4 байтам (32-бітні ОС) або 8 байтам (64-бітні ОС). Отже, навіть якщо різні об'єкти відрізняються за своїми розмірами (мають різний тип), адреси цих об'єктів завжди мають однаковий розмір. Наприклад, у 64-бітних застосунках змінна типу `char` має розмір 1 байт, змінна типу `int` – 4 байти, а змінна типу `double` – 8 байт, проте адреси всіх цих змінних будуть мати однакову довжину – 8 байт.

Так само як звичайні змінні, вказівники можна ініціалізувати в момент їх оголошення – тобто визначати їх. Для цього використовуються ті ж самі правила, що і для змінних, зокрема, правило співпадіння типів між змінною-вказівником та тим значенням, яке присвоюється цій змінній. Так, вказівнику типу `Тип*` можна присвоїти адресу (`Значення`) тільки змінної відповідного `Типу`. Наприклад, вказівник типу `int*` може вказувати лише на змінну типу `int`.

Для обчислення адреси деякого об'єкта в пам'яті використовується унарна *операція обчислення адреси*. В програмах мовою C/C++ вона позначається символом `&` (символ «комерційне і», «амперсанд»), який записується перед ідентифікатором об'єкта. Наприклад, якщо є змінна `int i`, результатом операції `&i` буде адреса цієї змінної `i`. Таким чином, для ініціалізації вказівника може використовуватись код, подібний до наведеного нижче:

```
int  i = 10; // Визначається змінна i.
int* p = &i; // Вказівник p вказує на i.
```

Тут вказівник `p` вказує на змінну `i`, в яку було записане значення `10`.

Досить часто виникає ситуація, коли деякий вказівник, який раніше вказував на одну змінну, потім має вже вказувати на іншу змінну того ж самого типу. Для реалізації такої операції слід змінити вміст вказівника – переініціалізувати його. Це виконується за тими ж правилами, що і переініціалізація змінних. Наприклад, розглянемо наступний фрагмент коду:

```
int  i = 10; // Визначається змінна i.
int* p = &i; // Вказівник p вказує на i.
int  j = 20; // Визначається змінна j.
p = &j;      // Вказівник p вже вказує на j.
```

Перші два рядки в цьому фрагменті ідентичні тим, що розглядались вище. У третьому рядку `int j = 20;` вводиться нова змінна `j`, ініцалізована значенням `20`. Після цього, вказівник `p` переініціалізується оператором `p = &j;` і

буде вже вказувати на нововведену змінну `j`.

При роботі з вказівниками використовується загальне правило-домовленість: *адреса існуючого об'єкта не може бути нульовою*. Тобто, якщо деякий вказівник ініціалізований значенням `0`, яке часто записується в програмах як макрос[1] `NULL`, це означає, що цей вказівник *не пов'язаний з реальним об'єктом* – він *ні на що не вказує*. Завдяки цьому можна легко розрізняти «нормальні» вказівники, які пов'язані з реально існуючими об'єктами в пам'яті, та *неприпустимі* (*нульові* або *пусті*) вказівники, які не пов'язані з жодним об'єктом.

Іншою особливістю вказівників є наявність т. зв. ***універсальних вказівників*** – вказівників типу `void*`. Такі вказівники можуть містити в собі адресу будь-якого об'єкту будь-якого типу. Вони *здатні вказувати на будь-що*, що знаходиться у пам'яті. Проте, перш ніж отримувати доступ до даних, на які вказує такий універсальний вказівник типу `void*`, цей вказівник потрібно конвертувати у вказівник на конкретний тип даних. Дійсно, для того, щоб працювати з даними, на які вказує вказівник, необхідно точно знати розмір і формат цих даних. А оскільки вказівник типу `void*` може вказувати на що завгодно, інформація про розмір і формат даних виявляється невизначеною. Тому для роботи з даними слід перейти від вказівника типу `void*` до вказівника конкретного типу за допомогою операції приведення типу. Приклад такого переходу наведено у наступному фрагменті коду:

```
void* p = 0; // Універсальний нульовий вказівник p.
int  i  = 10;// Визначається змінна i.
p = &i;      // Універсальний вказівник p вказує на i.
int* pi = (int*)p; // Приведення типу для вказівника p.
```

Спочатку вводиться універсальний вказівник `void* p`, який ініціалізується значенням `0`. Тобто введений вказівник `p` *може*, формально, *вказувати на будь-що*, але *поки що*, за рахунок ініціалізації нулем, *ні на що не вказує*. Далі вводиться змінна `i`, а її адреса записується у вказівник `p`. В останньому четвертому рядку коду вводиться новий вказівник `int* pi`, який за допомогою операції приведення типу `(int*)` ініціалізується адресою змінної `i`, яка була раніше збережена у вказівнику `p`.

Доступ до даних та їх модифікація через вказівник

За допомогою вказівника можна звернутись до того об'єкта в пам'яті, на який він вказує. Тим самим можна зчитувати або змінювати вміст об'єкта. Унарна операція доступу до об'єкта через вказівник називається ***операцією***

[1] Використання макросів буде розглядатись в лабораторній роботі № 6.

розіменування або ***розадресації*** вказівника. Залежно від типу об'єкта, в програмах на C/C++ вона може записуватись різним чином. Розглянемо найпростіший варіант її запису за допомогою символу «зірочки» `*`.

Якщо символ `*` записується безпосередньо перед вказівником, в програмі це інтерпретується як той об'єкт, на який вказує даний вказівник. Наприклад, якщо вказівник `int* p` вказує на змінну `int i`, операція `*p` інтерпретується як доступ до змінної `i`. Тобто фрагмент `*p` у програмі виявляється аналогом імені конкретної змінної `i`. Це дозволяє як зчитувати, так і змінювати вміст змінних та об'єктів не безпосередньо, а через вказівники на ці змінні та об'єкти. Наведемо приклад коду, який демонструє можливості такого підходу (лістинг 5.1).

Лістинг 5.1:

```
 1: #include <stdio.h>
 2:
 3: int main( void )
 4: {
 5:     int i = 1;// Визначається змінна i.
 6:     int j = 2;// Визначається змінна j.
 7:     void* pv = &i; // pv <- адреса i.
 8:     int*  pj = &j; // pj <- адреса j.
 9:
10:     printf("i = %d, j = %d\n", i, j);
11:
12:     // Змінюємо значення i через вказівник.
13:     *((int*)pv) = 3;
14:
15:     printf("i = %d, j = %d\n", i, j);
16:
17:     // Змінюємо значення j через вказівник.
18:     *pj = *pj * i; // j = j * i;
19:
20:     printf("i = %d, j = %d\n", *((int*)pv), *pj);
21:
22:     return 0;
23: }
```

У рядках 5, 6 визначаються змінні `i`, `j`, які ініціалізуються значеннями `1` та `2`. У рядку 7 вводиться універсальний вказівник `pv`, в який записується адреса змінної `i`. У рядку 8 створюється вказівник `pj` на змінну `j`. При виконанні коду рядка 10 на консоль виводиться текстове повідомлення i = 1, j = 2.

В рядку 13 у змінну `i`, на яку вказує вказівник `pv`, записується значення `3`. Для цього, спочатку, за допомогою фрагменту `(int*)pv` вказівник `pv`

конвертується у вказівник на змінну типу `int` (у вказівник типу `int*`). Потім отриманий вказівник типу `int*` розіменовується за допомогою операції `*`. Додаткові дужки навколо `(int*)pv` записані лише для більшої наочності коду. За необхідності їх можна опустити і код в рядку 13 записати так: `*(int*)pv = 3;`. З урахуванням коду рядка 13, функція `printf()` у рядку 15 виведе повідомлення: i = 3, j = 2.

У рядку 18 фрагмент `*pj` є еквівалентом безпосереднього звернення до змінної `j`. Отже, фактично, цей рядок реалізує операцію `j = j * i;`. При цьому `*pj` праворуч від знаку = (операції присвоєння) інтерпретується як зчитування даних зі змінної `j`, а `*pj` ліворуч від = означає запис у змінну `j` того значення, яке стоїть праворуч від =.

Виклик функції `printf()` у рядку 15 реалізується шляхом доступу до змінних `i` та `j` через вказівники на ці змінні.

Наприкінці наведемо ще одну, досить очевидну, проте важливу особливість використання вказівників. Якщо на деякий об'єкт в пам'яті вказує декілька вказівників, модифікація об'єкта через будь-який з цих вказівників автоматично означає, що всі вказівники будуть адресувати модифікований об'єкт.

Вказівники та масиви. Арифметика вказівників

У мові програмування C/C++ робота з масивами внутрішньо реалізується через роботу з вказівниками, отже, поняття масиву і вказівника виявляються суттєво пов'язаними між собою.

Ім'я масиву є вказівником на його початок – на його перший елемент з індексом 0. Тому наступний код є повністю коректним:

```
int  ar[3] = { 1, 2, 3 }; // Запис масиву ar.
int* p1    = ar;          // p1 <- адреса ar.
int* p2    = &ar[0];      // p2 <- адреса ar.
*p1 = 10;                 // ar[0] = 10.
*p2 = 5;                  // ar[0] = 5.
```

У цьому фрагменті коду, вказівники `p1` та `p2` вказують на один і той же елемент на початку масиву. Обидві форми запису, як `int* p1 = ar;`, так і `int* p2 = &ar[0];` є повністю еквівалентними, а відповідні вказівники містять одну і ту ж саму адресу.

Як відомо з лабораторної роботи № 4, для доступу до елементів масиву використовується операція індексування `[]`. Відповідний доступ, однак, можна отримати і за допомогою вказівників. Для цього слід врахувати, що зміна вказівника на +1 дозволяє перейти до наступного об'єкта того типу, на який вказує вказівник. А зміна вказівника на –1 дозволяє перейти до попереднього

об'єкта відповідного типу. Узагальнюючи це правило, одержимо, що зміна вказівника на ±*N* дозволяє перейти на *N* блоків вперед або назад відносно поточного блоку даних, на який вказує вказівник. Щоб проілюструвати дію цього правила, розглянемо масив `int ar[3]`, введений вище. Початково перший елемент цього масиву `ar[0]` містить число `1`, другий елемент `ar[1]` містить число `2`, а третій `ar[2]` – число `3`. Для доступу до елементу `ar[0]` може використовуватись вказівник `p1` або `p2`. Вказівники `p1+1` та `p2+1` будуть вказувати вже на другий елемент масиву `ar[1]`, а вказівники `p1+2` та `p2+2` – на третій елемент масиву `ar[2]`. Таким чином, для того, щоб отримати доступ до деякого елемента з індексом `Index` всередині масиву `ar`, необхідно спочатку визначити вказівник на цей елемент як `ar + Index` і після цього розіменувати отриманий вказівник: `*(ar + Index)`.

Важливою рисою арифметики вказівників є те, що зміна вказівника на деяку фіксовану величину ±*N* означає зсув в пам'яті на кількість байт ±*N*·`sizeof(Тип)`, де `Тип` – це той тип даних, на який вказує вказівник. Це є наслідком того, що зміна вказівника на ±*N* дозволяє перейти на *N* блоків вперед або назад відносно тої адреси блоку, яка зберігається у вказівнику. Щоб продемонструвати використання вказівників при роботі з масивами та підкреслити особливості арифметики вказівників розглянемо програму у лістингу 5.2.

Лістинг 5.2:

```
 1: #include <stdio.h>
 2:
 3: int main( void )
 4: {
 5:     // Вказівник на і як на набір символів.
 6:     int i = 123456;        // Має розмір 4 байти.
 7:     char* pc = (char*)&i; // і = масив символів.
 8:
 9:     printf("i == %d, [0] == %c, [1] == %c, [2] ==
10: %c, [3] == %c\n\n", i, *pc, *(pc+1), *(pc+2),
11: *(pc+3));
12:
13:     // Змінимо вміст і як набору символів.
14:     *pc++ = 'K';
15:     *pc++ = 'N';
16:     *pc++ = 'U';
17:     *pc   = '!';
18:
19:     pc = (char*)&i;
20:     printf("i == %d, [0] == %c, [1] == %c, [2] ==
21: %c, [3] == %c\n\n", i, *pc, *(pc + 1), *(pc + 2),
```

```
22:  *(pc + 3));
23:
24:      *(pc+ 3) = '\0';
25:      printf("i == %d, %s\n\n", i, pc);
26:
27:      return 0;
28:  }
```

У рядку 6 вводиться змінна `int i`, яка ініціалізується значенням `123456`. Ця змінна займає в пам'яті об'єм 4 байти, а значить її, формально, можна розглядати як масив з чотирьох символів типу `char`. Щоб працювати з таким штучно введеним масивом, у рядку 7 визначається вказівник на початок цього масиву: `char* pc = (char*)&i;`. Всі подальші операції з масивом реалізуються за допомогою вказівника `pc`.

Код у рядках 9-11 виводить за допомогою `printf()` значення змінної `i`, а також вміст пам'яті, де знаходиться `i`, як набору з чотирьох символів (див. перший рядок на рис. 22). Перший символ виводиться як значення `*pc`, другий – як `*(pc+1)`, третій – як `*(pc+2)`, і останній – як `*(pc+3)`. При цьому початковий вказівник `pc` не змінюється.

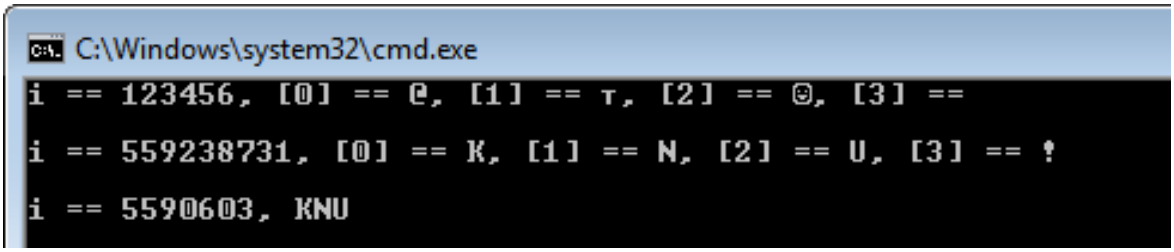


Рис. 22

У рядках 14-17 вміст масиву символів змінюється. Наприклад, код у рядку 14 `*pc++ = 'K';` спочатку зберігає в `*pc` – тобто в першому елементі символьного масиву – символ `'K'`. Після цього виконується операція постінкременту `pc++`, в результаті якої вказівник `pc` змінюється на +1 і починає вже вказувати на другий елемент масиву. Далі, в рядку 15, оператор `*pc++ = 'N';` зберігає в другому елементі масиву символ `'N'` і переходить (за рахунок постінкременту `pc++`) до наступного третього елементу. Цей же алгоритм повторюється в рядку 16 і без постінкременту `pc++` в рядку 17. В результаті при виконанні коду рядків 14-17, в область пам'яті, де розташована змінна `i`, послідовно записується набір символів `'K'`, `'N'`, `'U'`, `'!'`.

Для того, щоб вивести цю послідовність символів, у рядку 19 перевизначається вказівник `pc`. Він знову встановлюється як вказівник на початок символьного масиву – як адреса змінної `i`. Далі за допомогою `printf()` виводиться значення змінної `i`, а також вміст пам'яті, пов'язаної з `i`, у вигляді

набору з чотирьох символів (див. другий рядок на рис. 22).

Модифікація останнього символу в масиві відбувається у рядку 24 лістингу 5.2. Цей символ встановлюється у значення `'\0'`, що дозволяє інтерпретувати отриманий символьний масив як текстовий рядок в форматі C. Виклик `printf()` в рядку 25 виводить отриманий текстовий рядок `"KNU"` та еквівалентне йому представлення цілого числа (див. останній рядок на рис. 22).

Підкреслимо, що хоча між масивами та вказівниками є багато чого спільного, масиви та вказівники – це не одне і теж саме. Масив містить додаткову інформацію про кількість своїх елементів, яка відсутня у вказівника. Завдяки цьому, оператор `sizeof(масив)` завжди повертає правильний розмір масиву в байтах, в той час як оператор `sizeof(вказівник)` буде повертати 4 або 8 байт для відповідно 32-бітних або 64-бітних застосунків.

Операції з вказівниками виконуються трохи швидше, ніж операції індексації в масиві. Враховуючи це, у застосунках, де швидкодія є критично важливою, роботу з масивами часто реалізують через вказівники. Наведемо приклад того, як це можна реалізувати для випадку вибіркового сортування, яке розглядалось у лабораторній роботі № 4.

Лістинг 5.3:

```
 1: #include <stdio.h>
 2:
 3: int ar[7]  = { -1, 5, -10, 6, 3, 2, 41 };
 4:
 5: int arSize = sizeof(ar)/sizeof(ar[0]);
 6:
 7: int main( void )
 8: {
 9:     int minElem, tmp;
10:     int* p    = NULL;
11:     int* pj   = NULL;
12:     int* pMin = NULL;
13:
14: // Вивести вміст масиву ar.
15:     for( p = ar; p < ar+arSize; p++ )
16:         printf("%d ", *p);
17:
18: // Вибіркове сортування.
19:     // Проходимо масив в прямому напрямку.
20:     for( p = ar; p < ar + arSize-1; p++ )
21:     {
22:         minElem = *p;
23:         // Знаходимо найменший елемент у
24:         // підмасиві.
```

```
25:           for( pj = p+1; pj < ar + arSize; pj++ )
26:           {
27:               // Порівнюємо елементи.
28:               if( *pj < minElem )
29:               {
30:                   minElem = *pj;
31:                   pMin    = pj;
32:               }
33:           }
34:           // У попередньому циклі for знайдено
35:           // мінімальний елемент.
36:           if( p != pMin )
37:           {   // Встановити його на потрібне місце.
38:               tmp   = *p;
39:               *p    = minElem;
40:               *pMin = tmp;
41:           }
42:       }
43:
44:   // Вивести вміст масиву.
45:       printf("\n");
46:       for( p = ar; p < ar+arSize; p++ )
47:           printf("%d ", *p);
48:       printf("\n");
49:
50:       return 0;
51:   }
```

Програма з лістингу 5.3 є трохи довшою за аналогічну програму з лістингу 4.5, проте працює швидше. В оновленій версії програми не використовуються операції індексації масивів – всі операції виконуються безпосередньо за допомогою вказівників. Відповідно у програмі вже не потрібні змінні `i`, `j`, `minIndex`, які раніше використовувались як індекси елементів масиву у лістингу 4.5. Замість них введено три вказівники типу `int*`: `p`, `pj` та `pMin`. Перший з них (`p`) використовується для звичайного доступу до елементів масиву `ar`, другий (`pj`) – для доступу до елементів підмасиву всередині `ar`. Останній вказівник (`pMin`) вказує на мінімальний елемент всередині масиву.

Принцип роботи записаного коду ідентичний до коду, наведеного у лістингу 4.5. Наприклад, для виведення вмісту масиву `ar` на консоль раніше застосовувався код (див. рядки 12-13 у лістингу 4.5):

```
for( i = 0; i < arSize; i++ ) printf("%d ", ar[i]);
```

Тут за допомогою циклу `for` перебираються індекси `i` всіх елементів масиву `ar` і значення кожного елементу `ar[i]` виводиться на консоль за допомогою `printf()`. Аналогічний код з лістингу 5.3 виглядає так (рядки 15-16):

```
for( p = ar; p < ar+arSize; p++ ) printf("%d ", *p);
```

Видно, що перебір елементів у масиві йде через вказівник `p`. При цьому `p-ar`, по суті, є індексом елемента масиву, на який вказує `p`. Для доступу до елемента масиву використовується конструкція `*p`, а в кінці кожної ітерації циклу виконується операція `p++`, за рахунок якої вказівник `p` зсувається на +1 і починає вказувати на наступний елемент масиву `ar`.

Вказівники і функції

Параметри, які передаються в функцію, можуть бути вказівниками. Також функція може повертати величину, яка є вказівником довільного типу.

Параметри-вказівники, так само як і параметри інших типів, завжди передаються *за значенням*. Проте, використовуючи переданий вказівник, *можна змінити* той об'єкт, на який він вказує. Такий спосіб передачі параметрів називається передачею ***за вказівником***. Він є дуже зручним, коли всередині тіла функції необхідно змінити вміст змінної/об'єкта. Розглянемо приклад коду з функцією, яка має параметр-вказівник, модифікує за його допомогою зовнішній об'єкт, і потім повертає значення вказівника (лістинг 5.4).

Лістинг 5.4:

```
 1: #include <stdio.h>
 2:
 3: char s[] = "Glory_to_Ukraine!!!\n";
 4:
 5: char* ChangeString( char* _s )
 6: {
 7:     char* p = _s;  // Тимчасовий вказівник.
 8:     // Якщо *p = 0 - досягнуто кінця рядка.
 9:     while( *p )
10:     {
11:         if( *p == '_' ) *p = ' ';
12:         p++;
13:     }
14:     return _s;
15: }
16:
17: int main( void )
```

```
18:  {
19:      // Вивести вміст рядка.
20:      printf(s);
21:
22:      // Змінити вміст рядка і вивести його.
23:      printf(ChangeString(s));
24:
25:      return 0;
26:  }
```

У рядку 3 вводиться символьний масив `s`, який ініціалізується текстовим рядком `"Glory_to_Ukraine!!!\n"`. Форма запису `s[]` означає, що компілятор сам автоматично розраховує розмір масиву `s` так, щоб в ньому помістився текстовий рядок `"Glory_to_Ukraine!!!\n"`.

У рядках 5-15 визначається функція `ChangeString()`. В її тілі за допомогою циклу `while( *p )` перебираються усі символи `*p` в символьному масиві, на початок якого вказує переданий вказівник `char* _s`. Цикл виконується, поки не буде досягнутий нульовий символ `'\0'` з кодом `0`, який кодує кінець текстового рядка. В тілі циклу символ `'_'` замінюється на символ пробілу `' '`, після чого відбувається перехід до наступного символу за допомогою `p++`. Отже, функція виконує модифікацію текстового рядка і потім повертає вказівник на його початок за допомогою `return _s;`.

В тілі функції `main()` виклик `printf(s);` приводить до виводу тексту Glory_to_Ukraine!!! на консоль. У рядку 23 реалізується виклик внутрішньої функції `ChangeString(s)`. Вона змінює вміст рядка `s` і повертає вказівник на вже змінений рядок, який далі виводиться на екран за допомогою `printf()`. В результаті виводиться текст Glory to Ukraine!!!

Зазначимо, що при передачі функції параметра у вигляді масиву, цей масив автоматично конвертується у вказівник потрібного типу, і в функцію передається саме вказівник. А оскільки вказівник не містить інформації про розмір масиву, для коректного доступу до даних масиву, його розмір слід передавати в функцію додатковим параметром. Наприклад, наступна функція виконує модифікацію даних масиву, переданих через вказівник `ar` на початок масиву та його розмір `Size`:

```
void Modify( int* ar, int Size )
{
    for( int* p = ar; p < ar + Size; p++ )
        *p = *p * 2; // ar[i] *= 2;
}
```

Наведене правило типово ігнорують лише для текстових рядків в форматі C, в яких кінець масиву позначається за допомогою нульового символу `'\0'` (саме

за таким спрощеним принципом реалізовано функцію `ChangeString()` у лістингу 5.4).

Оскільки код довільної функції розміщується у пам'яті, можна ввести і *вказівник на функцію*. Синтаксис запису такого вказівника наступний:

```
Тип (*Ім'я_вказівника)(Аргументи_функції) = Значення;
```

Тут `Тип` – це тип результату, який має повертати функція.

`Ім'я_вказівника` – це ім'я *типу вказівника*, пов'язаного з потрібним класом функцій, які мають аргументи `Аргументи_функції` і повертають величину вказаного `Типу`. Ім'я типу вказівника має записуватись із зірочкою перед ним і має обов'язково включатись в круглі дужки.

`Аргументи_функції` – це перелік *типів* аргументів, які має приймати функція.

`Значення`, яким може ініціалізуватись вказівник на функцію, є іменем функції.

Для виклику функції через вказівник слід застосувати програмну конструкцію, в якій явно вказуються `Конкретні_аргументи`, що передаються функції:

```
(*Ім'я_вказівника)(Конкретні_аргументи);
```

Наприклад, для функції `char* ChangeString( char* _s )` з лістингу 5.4 введення та використання вказівника `pfn` на цю функцію може бути зроблене наступним чином:

```
char* (*pfn)(char*) = ChangeString;
(*pfn)(s);
```

У першому рядку вводиться вказівник `pfn` на деяку функцію, що приймає один параметр типу `char*` і повертає величину того ж типу `char*`. Вказівник відразу ж ініціалізується іменем функції `ChangeString`. Далі відбувається виклик функції `ChangeString` через вказівник `pfn`. При цьому фрагмент `(*pfn)` фактично виконує ту ж саму роль, що й ім'я функції `ChangeString`.

Посилання

Посилання (англ. reference) – це специфічний тип прихованого вказівника у мові C++, який при використанні завжди розіменовується і використовується як певний аналог (псевдонім) об'єкта, з яким він пов'язаний.

Синтаксис визначення посилання наступний:

```
Тип& Ім'я_посилання = Значення;
```

або

```
Тип &Ім'я_посилання = Значення;
```

`Тип` є типом об'єкта, з яким пов'язане посилання.

`Ім'я_посилання` є унікальним ідентифікатором – ім'ям відповідної змінної-посилання.

На відміну від вказівників, які можуть тільки оголошуватись, але не ініціалізуватись, посилання *має обов'язково ініціалізуватись* в момент свого введення і після цього змінюватись вже не може. Воно є аналогом вказівника, який обов'язково ініціалізується в момент свого створення, але в подальшому залишається незмінним. Ініціалізація посилання відбувається шляхом присвоювання йому `Значення` – імені об'єкта/змінної. Наприклад, код

```
int   i  = 3; // Створити змінну i; i = 3.
int& ri  = i; // ri - посилання на i.
int& ri2 = i; // ri2 - ще одне посилання на i.
```

визначає спочатку змінну `i`, а потім два посилання, `ri` та `ri2`, які обидва вказують на одну і ту ж саму змінну `i`.

Доступ до об'єкта, на яке вказує посилання, відбувається безпосередньо через запис його ідентифікатора. Додаткові символи, такі як `*` для операції розіменування вказівника, при роботі з посиланням не потрібні. При цьому компілятор самостійно інтерпретує записаний код певним чином, виходячи з його вигляду. Наприклад, операція `i = i * i + i;` через раніше введені посилання може бути записана наступним чином:

```
ri = ri * ri2 + ri;
```

Тут `ri` або `ri2` праворуч від знаку присвоювання `=` сприймаються як зчитування вмісту змінної, на яке вказує посилання (`ri` та `ri2` вказують на одну і ту ж саму змінну `i`). Операція `ri =` сприймається як запис в змінну `i`, пов'язану з посиланням `ri`, того значення, яке отримане в результаті обчислень `ri * ri2 + ri`.

Як вже зазначалось, переініціалізація посилань не дозволяється. Тому останній рядок у фрагменті коду

```
int  i  = 3;
int  j  = 5;
int& ri = i;
```

```
ri = j;
```

не буде інтерпретуватись як встановлення посилання `ri` на іншу змінну `j`. Він буде сприйматись компілятором як переініціалізація змінної `i` через посилання `ri`, тобто як код `i = j`.

Наведемо приклад функції, яка отримує параметри у вигляді посилань і повертає результат своєї роботи у вигляді посилання (лістинг 5.5).

Лістинг 5.5:

```
 1: #include <stdio.h>
 2:
 3: int& AddTo( int& a, const int& b )
 4: {
 5:     a += b;
 6:     return a;
 7: }
 8:
 9: int main( void )
10: {
11:     int i = 1;
12:     int j = 2;
13:
14:     printf("i = %d, j = %d\n", i, j);
15:
16:     AddTo(i, j);
17:
18:     printf("i = %d, j = %d\n", i, j);
19:
20:     return 0;
21: }
```

Функція `int& AddTo( int& a, const int& b )` виконує додавання двох аргументів (`i` та `j`) зі збереженням результату у змінній `i`, що пов'язана з першим аргументом (рядок 5). Оскільки її робота відбувається через посилання, функція може зчитувати вміст обох змінних і змінювати значення змінної `i`. Модифікатор `const` перед `int& b` вказує на те, що посилання `b` можна використовувати *тільки для зчитування значення змінної*. Модифікація змінної через *стале посилання*, записане з ключовим словом `const`, не дозволяється. Наприклад, код `b += a;` в тілі функції `AddTo()` компілюватись не буде, оскільки параметр `b` вважається сталим (`const`).

До виклику функції `AddTo()` в рядку 16, значення змінних `i` та `j` дорівнюють 1 та 2. Після виклику функції `AddTo()`, `i` набуває значення 3, а значення `j` не змінюється.

РОБОТА З ДИНАМІЧНОЮ ПАМ'ЯТТЮ

Для чого потрібна динамічна пам'ять і як з нею працювати

Мова C/C++ підтримує три основні способи виділення пам'яті:

Статичне виділення пам'яті використовується для статичних та глобальних змінних. Ця пам'ять виділяється під час запуску програми і залишається виділеною протягом всього часу її роботи. Після завершення роботи програми, статична пам'ять автоматично звільняється.

Локальне (автоматичне) виділення пам'яті застосовується для роботи з локальними змінними та для передачі параметрів у функції. Локальна пам'ять реалізується у вигляді стеку, який має змінний розмір, і то розширюється при вході в функцію, то стискається при виході з неї. При зміні розмірів стеку, та область пам'яті, яка опинилась за його межами, вважається заповненою невизначеними (недійсними) даними.

Динамічне виділенні пам'яті – це спосіб виділення пам'яті в адресному просторі програми за допомогою апаратних та програмних засобів, що контролюються ОС.

Сучасні процесори мають спеціальні регістри, команди та протоколи для адресації пам'яті, зв'язування віртуальної та фізичної пам'яті, переміщення та дефрагментації блоків динамічної пам'яті, їх захисту тощо. На основі цього, сучасні ОС реалізують ефективні універсальні менеджери пам'яті, здатні виконувати різні операції з блоками пам'яті, зокрема, виділяти та звільняти блоки динамічної пам'яті різного розміру.

На відміну від статичної та локальної пам'яті, які, зазвичай, мають обмежений об'єм (~ 1 МБ), динамічна пам'ять може виділятись блоками дуже великого об'єму (~ 1 ГБ і більше). До того ж, програма на C/C++ може самостійно контролювати процес виділення та звільнення блоків динамічної пам'яті, може контролювати доступ до цих блоків, змінювати їх розмір тощо. Це дає можливість працювати з блоками пам'яті, розмір яких задається і змінюється безпосередньо в процесі роботи програми (наприклад, залежно від даних, які вводить користувач). Отже, в програмах, які використовують динамічну пам'ять, можна зберігати дані майже довільного (як малого, так і великого) розміру, причому залежно від кількості (об'єму) даних, буде змінюватись і об'єм виділеної пам'яті. Це підвищує ефективність використання пам'яті програмою.

Динамічна пам'ять у C/C++ реалізована через доступ до великого сховища пам'яті, яке називається ***«купа»*** (англ. heap). У мовах програмування C та C++ цей доступ реалізований дещо різним чином, хоча принципи роботи з динамічною пам'яттю «купи» у C та C++ не відрізняються:

1) Спочатку, за допомогою засобів C або C++ виділяється блок динамічної пам'яті і програмі надається вказівник на початок цього блоку.

2) Використовуючи цей вказівник, програма заповнює блок пам'яті

даними, а також виконує будь-які інші потрібні маніпуляції з даними, що знаходяться всередині цього блоку.

3) Коли виділена пам'ять вже не потрібна, її звільняють, що робить вказівник на початок відповідного блоку пам'яті недійсним.

Будь-яка програма C/C++, яка використовує динамічну пам'ять, реалізує в своєму коді алгоритм, наведений вище. При цьому, кроки 1 та 3 є доволі однотипними для більшості програм – вони виконуються за допомогою існуючого коду стандартних бібліотек C/C++. Другий же крок є унікальним для кожної програми – він реалізується відповідно до тої задачі, яку має розв'язувати програма.

Робота з динамічною пам'яттю у C

Робота з динамічною пам'яттю у C реалізована за допомогою функцій, які стають доступними при підключенні файлів заголовків `stdlib.h` та/або `malloc.h`. Відповідних функцій близько двох десятків. Найбільш відомими з них є функції `malloc()`, `calloc()`, `realloc()` та `free()`.

Функція `malloc()` (скорочення від англ. memory allocate) виділяє пам'ять з «купи» і повертає вказівник на початок виділеного блоку. Синтаксис її використання наступний:

```
Вказівник = (Тип*)malloc(Розмір_блоку);
```

Ця функція виділяє пам'ять об'ємом `Розмір_блоку` у байтах і повертає вказівник типу `void*` на початок виділеного блоку. Далі цей вказівник типу `void*` потрібно перетворити у вказівник деякого типу `Тип*` і зберегти у змінній-вказівнику `Вказівник` (того ж самого `Типу*`) для подальшого використання.

Якщо виклик `malloc()` був успішним – тобто блок пам'яті потрібного розміру був виділений – `Вказівник` буде мати ненульове значення. Якщо ж вільної пам'яті недостатньо для виділення блоку потрібного розміру, `malloc()` поверне нульовий вказівник – тобто значення нуль (`NULL` чи `nullptr`), або згенерує виключення[1].

Наступний фрагмент коду виділяє за допомого `malloc()` пам'ять під змінну типу `double` та ще 12 байт пам'яті, доступ до яких можна отримати через вказівник `pc`:

```
double* pd = (double*)malloc(sizeof(double));
```

[1] Виключення C++ будуть розглядатись пізніше. Поки що ж, для спрощення, будемо вважати, що при недостачі пам'яті функція `malloc()` просто повертає значення нуль. Змінити поведінку `malloc()` можна за допомогою функції `_set_new_mode()`.

```
char* pc = (char*)malloc(12);
```

Оскільки розмір типу `char` дорівнює 1 байту, другий виділений блок пам'яті, на який вказує `pc`, можна розглядати як масив з 12 символів.

Якщо блок пам'яті виділяється під масив, що має містити деяку `Кількість` елементів певного `Типу`, функції `malloc()` слід передати повний розмір масиву в байтах, який зручно обчислити як `Кількість * sizeof(Тип)`. Відповідний виклик буде виглядати так:

```
Вказівник = (Тип*)malloc( Кількість * sizeof(Тип) );
```

Наприклад, якщо потрібно за допомогою `malloc()` виділити пам'ять під 49 елементів типу `int` – тобто створити в динамічній пам'яті масив з 49 цілих чисел, відповідний фрагмент коду буде мати вигляд:

```
int* p = (int*)malloc(49*sizeof(int));
```

Доступ до елементів такого масиву можна одержати через вказівник `p` на його початок.

Альтернативою виклику `malloc()` для масивів буде використання функції `calloc(num, size)`, якій передається кількість елементів у масиві `num` та розмір кожного елементу в байтах `size`. Як і `malloc()`, функція `calloc()` повертає вказівник типу `void*`, який, перш ніж використовуватись, має бути приведеним до вказівника на певний конкретний `Тип*`. Відмінністю між цими функціями є те, що `calloc()` додатково ініціалізує всі елементи масиву значенням `0`. Ось приклад використання функції `calloc()` для масиву чисел типу `float` з 10 елементами:

```
float* p = (float*)calloc(10, sizeof(float));
```

Змінити розмір вже виділеного блоку пам'яті можна за допомогою функції `realloc()`, яка має наступний прототип:

```
void* realloc( void* p, size_t size );
```

Тут `void* p` – це вказівник на раніше виділений блок пам'яті, отриманий при виклику функцій `malloc()`, `calloc()` або `realloc()`, `size_t size` – новий розмір блоку в байтах. Специфічний тип `size_t` є типом результату, який повертає оператор `sizeof()`. Цей тип, фактично, є аналогом типу `unsigned int`.

При зміні розміру блоку пам'яті за допомогою `realloc()`, вміст цього блоку пам'яті не змінюється. Тобто ті дані, які були збережені в блоці пам'яті,

на який вказує p, залишаться в межах цього блоку і після зміни його розмірів. Однак, фізичне розташування блоку пам'яті всередині «купи» може змінитись, а тому вказівник, який поверне realloc() може відрізнятись від вказівника p, який був переданий функції.

Якщо ж параметр p має значення NULL, функція realloc() працює так само, як і функція malloc() – вона виділяє новий блок пам'яті розміром size. Нижче наведені приклади використання функції realloc():

```
// Ідентично виклику malloc(10*sizeof(float)).
float* p1 = (float*)realloc(0, 10*sizeof(float));
// Розмір виділеного блоку пам'яті збільшується вдвічі.
float* p2 = (float*)realloc(p1, 20*sizeof(float));
```

Для звільнення раніше виділеного блоку динамічної пам'яті використовується функція free(). Вона звільняє раніше виділену пам'ять і повертає її назад до «купи». Прототип цієї функції має вигляд:

```
void free( void* p );
```

Параметром p, який передається free(), має бути коректно ініціалізований вказівник на блок динамічної пам'яті, інакше логічна структури «купи» може зазнати пошкоджень. Ось приклад коректного застосування функції free() для звільнення пам'яті, на яку вказує вказівник p:

```
if( p ){ free(p); p = NULL; }
```

Якщо вказівник p є дійсним (коректним, ненульовим, таким що вказує на реальний об'єкт у пам'яті) відбувається звільнення блоку пам'яті, на який вказує p за допомогою free(p);. Потім вказівник p встановлюється у значення нуль, тобто помічається як недійсний (такий, який ні на що не вказує). Цей останній оператор p = NULL;, формально, є не обов'язковим, проте його використання дозволяє значно покращити надійність коду для роботи з динамічною пам'яттю.

Іноді виникає потреба заповнити блок даних (нещодавно виділений або вже давно існуючий) певним конкретним значенням. Для цього зручно скористатись функцією memset(), яка має наступний прототип, оголошений у файлі memory.h:

```
void* memset( void* p, int s, size_t count );
```

Функція memset() заповнює блок даних, на який вказує p, символами кількістю count, які мають код символу s. Тобто встановлює перші count байт блоку даних у значення s. Після завершення роботи функція повертає сам

вказівник p. Наприклад, наступний код присвоює значення '\0' першим 100 байтам у блоці, на який вказує p, а потім записує у перші 10 байтів того ж блоку символ 'X':

```
memset(p, '\0', 100);
memset(p, 'X', 10);
```

Робота з даними, які зберігаються в блоках динамічної пам'яті, виконується за тими ж правилами, що і робота зі звичайними змінними та масивами. При цьому слід пам'ятати, що доступ до відповідних даних можна одержати тільки через вказівник, який повертають функції виділення динамічної пам'яті.

Розглянемо додаткові приклади використання функцій для роботи з динамічною пам'яттю (лістинг 5.6).

Лістинг 5.6:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <malloc.h>
 4: #include <memory.h> // Потрібно для memset().
 5:
 6: int main( void )
 7: {
 8: // Виділення пам'яті під змінну.
 9:     float* p = (float*)malloc(sizeof(float));
10:     if( p )
11:     {
12:         *p = 3.1415926535f;
13:
14:         printf("Pi == %f\n\n", *p);
15:
16:         free(p);
17:         p = NULL;
18:     }
19:     else printf("!ERROR: not enough memory!\n");
20:
21: // Виділення пам'яті під масив (malloc).
22:     size_t size1 = 3U;
23:     int*   p1 = (int*)malloc(size1*sizeof(int));
24:     if( p1 )
25:     {
26:         for( size_t i = 0; i < size1; i++ )
27:         {
28:             *p1 = (i+1);
```

```
29:             *p1 = *p1 * ((*p1)++);
30:             printf("*(p1+%d) == %d\n", i, *p1);
31:         }
32:
33:         free(p1);
34:         p1 = NULL;
35:     }
36:     else printf("!ERROR: not enough memory!\n");
37:
38:     printf("\n");
39:
40: // Виділення пам'яті під масив (calloc).
41:     size_t  size2 = 3U;
42:     double* p2 = (double*)calloc(size2,
43:         sizeof(double));
44:     if( p2 )
45:     {
46:         for( size_t i = 0; i < size2; i++ )
47:         {
48:             *(p2+i) = 1.0/double(i+1.0);
49:             printf("*(p2+%d) == %g\n",i,*(p2+i));
50:         }
51:
52:         printf("\n");
53:
54:         // Зміна розміру масиву (realloc).
55:         size_t size3 = size2*2;
56:         double* p3 = (double*)realloc(p2,
57:             size3*sizeof(double));
58:         if( p3 )
59:         {
60:             p2 = NULL; // Є доступ через p3.
61:             memset(p3+size2, 0, (size3-size2) *
62:                 sizeof(double));
63:             *(p3+size3-1) = double(size3);
64:
65:             for( size_t i = 0; i < size3; i++ )
66:             {
67:                 printf("*(p3+%d) == %g\n", i,
68:                     *(p3+i));
69:             }
70:             printf("\n\n");
71:         }
72:
73:         if( p2 )
```

```
74:           {
75:                free(p2);
76:                p2 = NULL;
77:           }
78:
79:           if( p3 )
80:           {
81:                free(p3);
82:                p3 = NULL;
83:           }
84:      }
85:      else printf("!ERROR: not enough memory!\n");
86:
87:      return 0;
88: }
```

У рядках 9-19 демонструється робота зі змінною типу `float`, яка зберігається у динамічній пам'яті.

Спочатку, у рядку 9, за допомогою `malloc()` виділяється блок даних потрібного розміру – розміру `sizeof(float)` байт. Вказівник на цей блок пам'яті приводиться до типу `float*` і зберігається як вказівник `p`.

Якщо отриманий вказівник `p` є коректним (ненульовим), виконується код рядків 12-17, інакше виводиться повідомлення про помилку `"!ERROR: not enough memory!\n"`.

Оператор `*p = 3.1415926535f;` (рядок 12) записує у змінну, що зберігається в динамічній пам'яті, наближене значення числа π. Після цього виводиться значення цього числа на консоль (рядок 14, рис. 23). У рядках 16-17 раніше виділений блок динамічної пам'яті звільняється за допомогою `free()`, а вказівник на цей блок пам'яті `p` позначається як недійсний (нульовий).

Виділення пам'яті під масив за допомогою `malloc()` демонструється у фрагменті коду, що займає рядки 22-36.

У рядку 22 визначається змінна `size_t size1`, що дорівнює потрібній кількості елементів у масиві. На її основі розраховується розмір масиву в байтах `size1*sizeof(int)` і це розраховане значення передається функції `malloc()`, яка викликається в рядку 23. Вказівник на виділений блок пам'яті (на масив) зберігається у вказівнику `p1`.

Якщо отриманий вказівник `p1` є дійсним (ненульовим), виконується код рядків 26-34, інакше виводиться повідомлення про помилку.

У циклі `for` (рядки 26-31) відбувається ініціалізація елементів масиву, що розміщуються в динамічній пам'яті, на яку вказує `p1`:

```
*p1 = (i+1);
```

```
*p1 = *p1 * ((*p1)++);
```

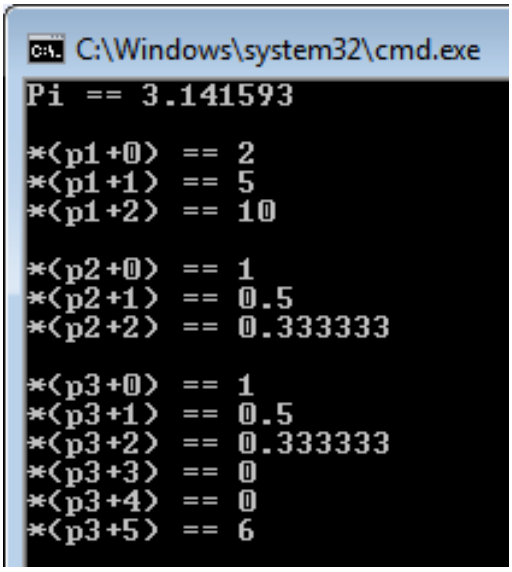


Рис. 23

Далі, значення вже ініціалізованих елементів масиву виводяться на консоль за допомогою `printf("*(p1+%d) == %d\n", i, *p1);` (рис. 23).

Звільнення динамічної пам'яті, пов'язаної з масивом, відбувається у рядку 33 через виклик `free(p1)`. Потім вказівник `p1` позначається як недійсний.

Найбільш складний фрагмент коду програми записано в рядках 41-85. У цьому коді спочатку за допомогою `calloc()` виділяється блок динамічної пам'яті під масив з `size2` чисел типу `double` (рядки 41-42). Якщо виклик `calloc()` був успішним, масив заповнюється даними, які потім виводяться на консоль (рядки 46-50, див. рис. 23).

У рядках 55-57 проводиться зміна розміру раніше виділеного блоку пам'яті за допомогою `realloc()`. Якщо виклик `realloc()` був успішним, кількість елементів у масиві збільшується (з `size2` до `size3`). Коректним вказівником на початок цього масиву буде вже `p3`, а не `p2`, тому у рядку 60 `p2` ініціалізується значенням нуль, щоб підкреслити те, що цей вказівник вже є недійсним.

У рядках 61-62 відбувається ініціалізація даних, пов'язаних з додатковими елементами масиву, які з'явились внаслідок розширення блоку пам'яті. Це відбувається за допомогою виклику `memset(p3+size2, 0, (size3-size2) * sizeof(double));`. Адреса `p3+size2` є початковою адресою блоку пам'яті, який був доданий при виклику `realloc()`. Величина `(size3-size2) * sizeof(double)` визначає об'єм у байтах доданого блоку. Усі байти всередині доданого блоку пам'яті заповнюються значенням `0`, що відповідає дійсним числам `0.0`. Потім у рядку 63 виконується оператор `*(p3+size3-1) = double(size3);`, що ініціалізує останній елемент масиву значенням `size3`.

Використовуючи вказівник `p3`, вміст розширеного масиву виводиться на консоль (рядки 65-69, рис. 23). Як видно з останнього фрагменту даних на рис. 23, при розширенні блоку пам'яті за допомогою `realloc()`, данні, які раніше містились у цьому блоці (масиві чисел типу `double`) зберігаються.

Завершення роботи з блоками динамічної пам'яті відбувається у рядках 73-83. Раніше виділені блоки звільняються за допомогою `free()`, а відповідні вказівники позначаються як недійсні.

Робота з динамічною пам'яттю у C++

Блоки динамічної пам'яті у C++ виділяються не за допомогою функцій[1], а спеціальних операторів `new` та `new[]`. Принцип їх роботи схожий з принципом роботи функцій `malloc()` або `calloc()`.

Оператор `new` виділяє блок пам'яті для збереження *одного єдиного* об'єкта вказаного типу. Він повертає вказівник на виділений блок пам'яті, який відповідає вказаному типу об'єкта. Синтаксис використання оператора `new` наступний:

```
Вказівник = new Тип;
```

Наприклад, виділення пам'яті для змінних типу `double` та типу `int` може використовуватись код наступного вигляду:

```
double* pd = new double;
int*    pi = new int;
```

Можна також використовувати форму оператора `new`, яка дозволяє виконувати ініціалізацію новоствореної змінної. Для цього, після типу змінної в круглих дужках вказується значення, яке буде використовуватись для її ініціалізації. Наприклад, код `int* pi = new int(3);` виділить пам'ять під змінну типу `int` і ініціалізує її значенням `3`.

Оператор `new[]` використовується для виділення динамічної пам'яті під *масив* з вказаною `Кількістю` елементів певного `Типу`. Синтаксис використання цього оператора наступний:

```
Вказівник = new Тип[Кількість];
```

Наприклад, фрагмент коду

```
int* p = new int[100];
```

виділяє блок пам'яті, достатній для збереження масиву з 100 цілих чисел. Якщо деяка змінна `unsigned int u` містить число потрібних елементів у масиві, код для виділення динамічної пам'яті під цей масив буде мати вигляд:

---

[1] В програмах на C++ можна вільно користуватись функціями `malloc()`, `calloc()`, `realloc()` та `free()`, які обговорювались раніше. Проте використання операторів C++ для виділення та звільнення динамічної пам'яті дає значні переваги при роботі зі структурованими типами даних, в першу чергу, класами, які будуть розглядатись пізніше. Тому в програмах на C++ рекомендується виділяти і звільняти блоки динамічної пам'яті за допомогою операторів C++.

```
int* p = new int[u];
```

Для звільнення динамічної пам'яті, виділеної оператором `new` застосовується оператор `delete`. Якщо ж пам'ять виділялась за допомогою оператора `new[]`, вона має звільнятись оператором `delete[]`. Обидва ці оператори мають однотипний синтаксис використання:

```
delete Вказівник;
delete[] Вказівник;
```

де `Вказівник` – це вказівник, який повертається операторами `new` або `new[]` при виділенні пам'яті. Наприклад, якщо `p` – це вказівник на змінну, що розташована у динамічній пам'яті, а `pa` – вказівник на масив, звільнення виділеної пам'яті, пов'язаної з `p` та `pa`, має відбуватись за допомогою операторів:

```
delete p;
delete[] pa;
```

Зазначимо декілька особливостей використання операторів `new`, `new[]`, `delete` та `delete[]`:

1) На відміну від функцій `malloc()`, `calloc()`, `realloc()`, які повертають вказівник типу `void*`, оператори `new` та `new[]` відразу ж повертають вказівник потрібного типу. Отже, при використанні `new` та `new[]` операція приведення типу вказівника не потрібна.

2) У мові C++ немає оператора для перевиділення динамічної пам'яті, який працював би подібно функції `realloc()`. Відповідні операції слід емулювати шляхом використання операторів `new`, `new[]`, `delete` та `delete[]` (див. лістинг 5.7).

3) За неможливості виділити потрібний об'єм динамічної пам'яті оператори `new` та `new[]` за замовчуванням генерують *виключну ситуацію*[1]. Генерація виключної ситуації перериває нормальний порядок виконання програми. Отже, код, який просто перевіряє коректність вказівника, що повертає `new` або `new[]`, шляхом порівняння його з нулем, може не спрацювати належним чином. Цю поведінку операторів `new`, `new[]` можна змінити викликавши функції `_set_new_mode()` та `_set_new_handler()`.

4) Пам'ять, яка виділялась за допомогою `new`, повинна звільнятись за допомогою `delete`, а пам'ять, виділена `new[]` – за допомогою `delete[]`. Порушення цього правила не гарантує коректної роботи операторів `new`, `new[]`, `delete` та `delete[]`.

5) Також не рекомендується при роботі *з одним і тим же самим* блоком

[1] Використання виключних ситуацій буде розглядатись пізніше.

динамічної пам'яті змішувати способи виділення і звільнення пам'яті, характерні для мов C та C++. Наприклад, не рекомендується виділяти пам'ять за допомогою `new` і потім звільняти її за допомогою `free()`. У той же час в одній програмі можна використовувати обидва підходи для роботи с динамічною пам'яттю, якщо для її виділення та звільнення використовується один і той же самий механізм, характерний або для мови C, або для мови C++.

На завершення, наведемо приклад програми, аналогічної до програми з лістингу 5.6. Вона реалізує виділення та звільнення динамічної пам'яті за допомогою операторів `new`, `new[]`, `delete` та `delete[]`.

Лістинг 5.7:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <malloc.h>
 4: #include <memory.h> // Потрібно для memset().
 5:
 6: int main( void )
 7: {
 8: // Виділення пам'яті під змінну (new).
 9:     float* p = new float;
10:     if( p )
11:     {
12:         *p = 3.1415926535f;
13:
14:         printf("Pi == %f\n\n", *p);
15:
16:         delete p;
17:         p = NULL;
18:     }
19:     else printf("!ERROR: not enough memory!\n");
20:
21: // Виділення пам'яті під масив (new[]).
22:     size_t size1 = 3U;
23:     int*   p1    = new int[size1];
24:     if( p1 )
25:     {
26:         for( size_t i = 0; i < size1; i++ )
27:         {
28:             *p1 = (i+1);
29:             *p1 = *p1 * ((*p1)++);
30:             printf("*(p1+%d) == %d\n", i, *p1);
31:         }
32:
33:         delete[] p1;
```

```
34:          p1 = NULL;
35:      }
36:      else printf("!ERROR: not enough memory!\n");
37:
38:      printf("\n");
39:
40:  // Виділення пам'яті під масив (new[]).
41:      size_t  size2 = 3U;
42:      double* p2    = new double[size2];
43:      if( p2 )
44:      {
45:          for( size_t i = 0; i < size2; i++ )
46:          {
47:              *(p2+i) = 1.0/double(i+1.0);
48:              printf("*(p2+%d) == %g\n", i,
49:                  *(p2+i));
50:          }
51:
52:          printf("\n");
53:
54:          // Зміна розміру масиву (new[]/delete[]).
55:          size_t size3 = size2*2;
56:          double* p3 = new double[size3];
57:          if( p3 )
58:          {
59:              // Скопіювати дані.
60:              memcpy(p3, p2, size2*sizeof(double));
61:              // Звільнити попередній масив.
62:              delete[] p2;
63:              p2 = NULL;
64:              // Ініціалізувати додаткові елементи.
65:              memset(p3+size2, 0, (size3-size2) *
66:                  sizeof(double));
67:              *(p3+size3-1) = double(size3);
68:
69:              for( size_t i = 0; i < size3; i++ )
70:              {
71:                  printf("*(p3+%d) == %g\n", i,
72:                      *(p3+i));
73:              }
74:              printf("\n\n");
75:          }
76:
77:          if( p2 )
78:          {
```

```
79:                 delete[] p2;
80:                 p2 = NULL;
81:             }
82:
83:             if( p3 )
84:             {
85:                 delete[] p3;
86:                 p3 = NULL;
87:             }
88:         }
89:         else printf("!ERROR: not enough memory!\n");
90:
91:         return 0;
92:  }
```

У рядку 9 замість виклику `malloc()` використовується оператор `new`, а у рядку 16 замість `free()` використовується `delete`. Аналогічно у рядках 23 та 42 замість `malloc()` та `calloc()` записується оператор `new[]`. Виділена ним динамічна пам'ять звільняється за допомогою `delete[]`.

Найбільші зміни в програмі відбуваються у рядках 56-63. Раніше у цьому місці програми використовувалась функція `realloc()`, яка збільшувала розмір пам'яті, відведений під масив. Оскільки у C++ немає аналогічного оператора, дію `realloc()` слід емулювати за допомогою операторів `new[]` та `delete[]`. Для цього спочатку за допомогою `double* p3 = new double[size3];` (рядок 56) виділяється новий масив потрібного розміру. При цьому раніше виділений масив, пов'язаний з вказівником `p2`, продовжує зберігатись у динамічній пам'яті. Наступного кроку дані з масиву, на якій вказує `p2`, переносяться у новий масив, на який вказує `p3` (рядок 60). Це відбувається шляхом виклику стандартної функції `memcpy()`, яка копіює в `p3` з `p2` кількість байт, що дорівнює `size2*sizeof(double)`. Тепер, коли раніше існуючі дані вже збережені, масив, на який вказує `p2`, більше не потрібен, тому він звільняється за допомогою `delete[] p2;`, а відповідний вказівник позначається як недійсний (`p2 = NULL;`).

Інші фрагменти програми з лістингу 5.7 нічим не відрізняються від аналогічних фрагментів у лістингу 5.6. Результат роботи обох програм є однаковим (див. рис. 23).

Завдання

1) Реалізуйте завдання з лабораторної роботи № 4, що відповідають вашому номеру варіанта, використовуючи операції з вказівниками замість

операцій індексації масивів.

2) Напишіть функції `AddTo()`, `SubTo()`, які виконують операції `x += y`, `x -= y`, зі змінними `x` та `y` типу `int`. Параметри мають передаватись у функції через посилання. Функції мають повертати посилання на модифіковану величину `x`. Використовуючи для `x` значення вашого номера варіанту з лабораторної роботи № 4, а для y – довільне число в межах від 0 до 20, яке вводиться з клавіатури, розрахувати і вивести значення `x` після виконання коду `AddTo(AddTo(x, y), SubTo(x, AddTo(y, x)))`.
3) Введіть вказівники на функції `AddTo()` та `SubTo()` з п. 2, і реалізуйте код п. 2 за допомогою виклику цих функцій через вказівники.
4) Модифікуйте програму, написану в п. 1 так, щоб вона працювала не зі статичними масивами, а з масивами, які розміщуються у динамічній пам'яті. Напишіть дві версії програми: в одній для виділення/звільнення динамічної пам'яті мають використовуватись функції стандартної бібліотеки C, в іншій – оператори C++.
5) Створіть функцію `ReadAndPrintInversed()`, яка має працювати за наступним алгоритмом: 1) у динамічній пам'яті виділяється символьний масив великого розміру, здатний зберігати 1024 символи, включаючи `'\0'`; 2) у цей масив зчитується текстовий рядок з консолі; 3) розмір символьного масиву корегується так, щоб у ньому було місце тільки для збереження введеного рядка; 4) виконується інвертування рядка (останній символ має помінятись місцем з першим, другий – з передостаннім тощо); 5) модифікований рядок виводиться на консоль; 6) виділена раніше пам'ять звільняється і функція завершує свою роботу.

Контрольні запитання

1. Що таке вказівник? Для чого потрібні вказівники?
2. Який синтаксис оголошення та визначення вказівника? Як відбувається ініціалізація та переініціалізація вказівників?
3. Що таке вказівник типу `void*`? Що таке нульовий вказівник? Наведіть приклади застосування таких вказівників у програмах.
4. Яким чином можна зчитати або модифікувати дані, на які вказує вказівник? Наведіть приклади.
5. Що спільного і в чому різниця між масивами та вказівниками? Чи можна використовувати вказівники для доступу до елементів масиву? Наведіть приклади.
6. Поясніть правила арифметики вказівників. Наведіть приклади як ці правила можна застосувати на практиці.
7. Чи можуть функції працювати з вказівниками? Приймати параметри у вигляді вказівників? Повертати значення вказівника? Наведіть приклади.

8. Що таке вказівник на функцію? Наведіть приклад.
9. Що таке посилання? У чому відмінність між вказівниками та посиланнями? Наведіть приклади.
10. Який синтаксис визначення посилань? Чи можна посилання не визначати, а оголошувати? Виконувати переініціалізацію посилань?
11. Чи можуть функції працювати з посиланнями? Приймати параметри у вигляді посилань? Повертати значення посилання? Наведіть приклади.
12. Для чого потрібна динамічна пам’ять? Який загальний принцип роботи з блоками динамічної пам’яті?
13. Поясніть яким чином відбувається робота з динамічною пам’яттю за допомогою функцій стандартної бібліотеки C. Наведіть приклади.
14. Поясніть яким чином відбувається робота з динамічною пам’яттю за допомогою операторів C++. Наведіть приклади.

Лабораторна робота № 6.
Директиви препроцесора C/C++. Побітові операції. Робота з файлами

Мета роботи:

1) Ознайомитись з директивами препроцесора. Навчитись створювати і застосовувати макроси.
2) Навчитись використовувати побітові операції.
3) Ознайомитись з особливостями роботи з бінарними і текстовими файлами. Навчитись зчитувати дані з файлів та записувати дані у файли за допомогою функцій стандартної бібліотеки C.

ДИРЕКТИВИ ПРЕПРОЦЕСОРА

Препроцесор C/C++ – це програмний інструмент, який генерує на основі вхідного .c/.cpp файлу проміжний (модернізований) варіант тексту програми, який далі обробляється компілятором C/C++. Цей модернізований текст програми може як розташовуватись в динамічній пам'яті (це пришвидшує процес компіляції), так і зберігатись у файлі на диску.

Обробка будь-якого .c/.cpp файлу обов'язково починається з його аналізу препроцесором C/C++. На цьому етапі, використовуючи спеціальну «внутрішню мову програмування» – *директиви препроцесора*, можна суттєво змінити логіку роботи програми. Крім того, використання директив препроцесора може помітно спростити вигляд тексту програми на C/C++, зробити цю програму більш універсальною, пришвидшити її компіляцію тощо. Тому розуміння принципів роботи препроцесора і знання його директив важливі для професійного написання програм мовами C/C++.

Директиви препроцесора – це команди, записані в тексті програми мовою C/C++, які виконуються до початку її трансляції в об'єктний код. Директиви препроцесора дозволяють змінювати сприйняття компілятором тексту програми без зміни самого цього тексту. Наприклад, за допомогою директив препроцесора можна забороняти компіляцію частин тексту програми.

Згідно синтаксису мов C/C++ усі директиви препроцесора починаються зі спеціального символу `#` (символ діеза або решітки). За цим символом безпосередньо записується потрібна директива, наприклад, `#include`, а також, якщо потрібно, аргумент або аргументи директиви. Якщо директива настільки довга, що не вміщується в рядок, її можна розбивати на кілька рядків, які розділяються символом зворотного слеша `\`. В кінці директиви, на відміну від операторів C/C++, крапка з комою `;` не ставиться.

Загальний формат синтаксису директив препроцесора має вигляд:

```
#Ім'я_директиви Аргумент(и)
```

`Ім'я_директиви` є наперед визначеним іменем з набору директив препроцесора. Це ім'я сприймається як зарезервоване слово мови C/C++.

`Аргумент(и)` – будь-які параметр або параметри, що дозволені для використання із записаною директивою.

Директиви препроцесора можна умовно розділити на кілька груп за характером їх застосування:

Директиви `#if`, `#ifndef`, `#elif`, `#ifdef`, `#else`, `#endif` забезпечують виконання умовної компіляції, яка використовується для отримання декількох варіантів однієї й тієї ж програми.

Директива `#include` дає можливість підключати файли заголовків.

Директиви `#define` та `#undef` дозволяють створювати та видаляти *макроси* (*псевдофункції*), що іноді важливо для гнучкої роботи з кодом.

Директиви `#pragma`, `#error`, `#line` тощо дозволяють отримати доступ до внутрішньої інформації компілятора.

Коротко розглянемо деякі з перелічених директив.

#define

Директива використовується для створення символьних констант або *макросів* (псевдофункцій). У найпростішому випадку вона має синтаксис:

```
#define text replacement
```

Згідно цієї директиви текстовий фрагмент `text` у всьому файлі буде замінено на текстовий фрагмент `replacement`, який, як правило, пов'язаний із константою або виразом. Наприклад, рядки

```
#define M_Pi    3.141592653589
#define C_Null '\0'
```

пов'язують число `3.141592653589` та символ `'\0'` з текстовими іменами `M_Pi` та `C_Null`. Це дозволяє в тексті програми всюди використовувати текстові фрагменти `M_Pi` та `C_Null` замість `3.141592653589` та `'\0'`, що робить програму легшою для сприйняття. До того ж, якщо виникне потреба змінити вміст макросів `M_Pi` та `C_Null`, це достатньо буде зробити лише в тих рядках, де вони визначені за допомогою `#define`.

Окрім того, що директива `#define` пов'язує між собою два текстові фрагменти, вона також робить ***визначеним*** (заданим, відомим, введеним)

вказаний після `#define` текстовий ідентифікатор – символьну константу `text`. В подальшому це можна використати для перевірки того, визначений цей ідентифікатор в певному місці програми, чи ні, що дозволяє керувати процесом компіляції програми – виконувати її умовну компіляцію. Для визначення текстового ідентифікатора (символьної константи) `Id` достатньо написати директиву `#define`, в якій після `Id` немає нічого або є тільки пробіли:

```
#define Id
```

Директиву `#define` також можна використати для створення *макрофункції* або *макроса* (*псевдофункції*). Такий ***макрос*** – це набір операцій C/C++ (фрагмент коду програми), який може виконуватись довільну кількість раз. Синтаксис запису макроса наступний:

```
#define macro(arg) replacement
```

Цей запис означає, що замість текстового фрагменту `macro(arg)` з поки що довільним аргументом `arg` в текст програми буде підставлятись фрагмент `replacement`. Отже, записаний макрос визначає загальне правило перетворення текстового фрагменту `macro(arg)` у фрагмент `replacement`. Проте кінцевий результат цього перетворення – фрагмент тексту програми на C/C++ – буде залежати від того, який саме аргумент `arg` буде передано макросу. При виклику макроса, в його тіло підставляється конкретний переданий параметр `arg` (або декілька параметрів), що дуже нагадує передачу одного або кількох аргументів функції. Однак, на відміну від функції, результатом виконання макроса (результатом його обробки препроцесором) є не об'єктний код, а фрагмент тексту програми на C/C++. Розглянемо ряд прикладів:

```
#define SQR(x)          (x*x)
#define Circle_Len(r) 2.0*M_Pi*r
#define VMOD(a, b)      sqrt(a*a+b*b)
```

Макрос `SQR(x)` дозволяє обрахувати квадрат величини `x`. Макрос `Circle_Len(r)` розраховує довжину кола з радіусом `r`, використовуючи раніше визначену макроконстанту `M_Pi`. Макрос `VMOD(a, b)` розраховує модуль двовимірного вектора з компонентами `a` та `b`.

Як видно, при використанні макросів зовсім не зазначаються типи змінних. Це відбувається тому, що препроцесор при обробці макроса не генерує об'єктний код. Він лише заміняє один фрагмент тексту програми іншим. Відповідно коректність отриманого коду (з точки зору правил мови C/C++) перевіряється не препроцесором, а компілятором, який буде аналізувати модифікований текст програми, згенерований препроцесором.

Незважаючи на зручність макросів, слід пам'ятати, що їх використання

не завжди є безпечним. Наприклад, код `SQR(x+y)` буде трансльований препроцесором у вираз `(x+y*x+y)`, який, очевидно, не співпадає з очікуваним результатом `(x+y)*(x+y)`. Тому, щоб уникнути цієї помилки, наведений вище макрос слід переписати так:

```
#define SQR(x)          ((x)*(x))
```

Використання таким чином введеного макроса дасть правильний результат для `SQR(x+y)`. Проте, навіть, цей, вже оновлений, макрос може призвести до появи невірного коду з точки зору логіки роботи програми, якщо йому передати аргументом величину вигляду `i++` або `--i`.

#if, #elif, #else, #endif

За характером свого використання ці директиви схожі до умовних операторів `if` або `if-else`. Відмінність між ними полягає в тому, що директиви препроцесора перевіряють виконання тієї чи іншої умови *лише під час компіляції програми*, а оператори `if`, `if-else` – виконують цю перевірку під час виконання програми.

В основному директиви `#if`, `#elif`, `#else`, `#endif` використовуються в двох випадках:

- Для виконання ***умовної компіляції*** – компіляції частин тексту програми лише за виконання певних умов. Це виявляється особливо корисно для створення різних версій однієї і тієї ж програми.
- Для коментування фрагментів програми. Наприклад, заключаючи певний блок коду в «дужки» `#if 0` та `#endif`, можна його «закоментувати» (виключити з розгляду компілятором).

Синтаксис використання директив `#if`, `#elif`, `#else`, `#endif` наступний (обов'язковими елементами є лише директиви `#if` та `#endif`):

```
#if Val_if
    Block_if
#elif Val_elif_1
    Block_elif_1
…
#elif Val_elif_N
    Block_elif_N
#else
    Block_else
#endif
```

В момент початку компіляції препроцесор перевіряє за допомогою `#if`

коректність (не рівність нулеві) величини Val_if. Якщо це так, то в тексті програми враховується (залишається для обробки компілятором) лише блок коду Block_if, а інші блоки Block_elif_1, …, Block_elif_N та Block_else ігноруються.

Якщо ж величина Val_if є хибною (нульовою), виконується перевірка інших значень Val_elif_1, Val_elif_2, …, та Val_elif_N. Якщо одне із цих значень виявляється істинним, в програмі залишається тільки той блок коду, що пов'язаний з цим значенням.

Якщо ж жодна з величин Val_if, Val_elif_1, Val_elif_2, …, Val_elif_N не є істинною, в програмі залишається останній блок коду Block_else.

Команда #endif позначає кінець складної директиви #if-#endif.

Наступний код демонструє приклад використання директиви #if-#endif:

```
#if 1
    int x = 1;
#else
    float x = 1.0;
#endif
```

У першому рядку #if 1 величина 1 є істинною, тому в програмі буде враховуватись лише фрагмент коду int x = 1;. Фрагмент float x = 1.0; буде при цьому ігноруватись.

Якщо ж в першому рядку поміняти 1 на 0, фрагмент int x = 1;, навпаки, буде ігноруватись, а враховуватись в програмі буде оператор float x = 1.0;. Тобто змінюючи константу, задану при #if (або #elif) можна коментувати фрагменти тексту програми і керувати процесом її компіляції.

Досить часто директива #if-#endif використовується для перевірки того, визначена певна символьна константа, чи ні. Це відбувається за допомогою додаткової команди препроцесора defined(s), яка повертає 1, якщо символьна константа визначена, і 0 в іншому випадку. При використанні defined можна також використовувати логічні операції І, АБО, НЕ, які кодуються таким же самим чином, як і в операторах C/C++ (див. лабораторну роботу № 2). Розглянемо наступний приклад:

```
#if defined(UNICODE) && defined(CHAR32)
    #define tCHAR    int
#elif defined(UNICODE) && !defined(CHAR32)
    #define tCHAR    wchar_t
#else
#define tCHAR    char
#endif
```

В умові `#if defined(UNICODE) && defined(CHAR32)` перевіряється чи визначені символьні константи (часто говорять, просто символи) `UNICODE` та `CHAR32`. Якщо це так, то `tCHAR` визначається як `int`.

Якщо ж виконуються умови `#elif defined(UNICODE) && !defined(CHAR32)`, тобто символ `UNICODE` визначений, а `CHAR32` – ні, `tCHAR` визначається як `wchar_t`.

В іншому випадку (блок `#else`) `tCHAR` визначається як стандартний тип `char`.

#ifdef, #ifndef

Директиви `#ifdef` та `#ifndef`, як правило, використовують для умовної компіляції коду. Директива `#ifdef s` є тотожною директиві `#if defined(s)`, а `#ifndef s` є скороченою формою запису для `#if !defined(s)`. Відповідно, синтаксис використання цих директив є наступним:

```
#ifdef s
    Block_1
#endif

#ifndef s
    Block_2
#endif
```

Якщо символ `s` *визначений* за допомогою `#define`, код, що відповідає блоку `Block_1`, буде враховуватись при компіляції програми, а код блоку `Block_2` – навпаки, буде ігноруватись. Якщо ж символ `s` є невизначеним, компілятор буде обробляти код блоку `Block_2` і буде ігнорувати код блоку `Block_1`.

Ці директиви корисні, коли потрібно скомпілювати той чи інший фрагмент коду програми залежно від значення, яке може бути або істиною (не нуль, `true`) або хибою (нуль, `false`). Наприклад, вміст майже всього файлу stdio.h зі стандартної бібліотеки C, включений у наступний блок `#ifndef-#endif`:

```
#ifndef _INC_STDIO
#define _INC_STDIO
// Вміст файла...
#endif
```

Наявність цього блоку означає, що доступ до вмісту файлу stdio.h буде відбуватись тільки тоді, коли ще не визначений символ _INC_STDIO. Цей символ визначається у рядку, наступному після #ifndef _INC_STDIO, за допомогою #define _INC_STDIO. Отже, вміст файлу stdio.h буде доступним, тільки якщо цей файл ще не оброблявся препроцесором (тобто директива #define _INC_STDIO ще не виконана). Якщо ж файл вже оброблявся і символ _INC_STDIO є визначеним, все що знаходиться між #ifndef _INC_STDIO та #endif у подальшому буде ігноруватись. Наведений приклад показує як досягнути того, щоб файл заголовку оброблявся компілятором лише один раз.

#include

Директива #include дозволяє включити у файл (типово .c/.cpp) вміст іншого файлу (як правило, файлу заголовку). Вона найчастіше використовується у будь-яких програмах мовою C/C++ і має наступний синтаксис:

```
#include <file>
#include "file"
```

Якщо у директиві #include використовується форма запису <file>, пошук файлу file буде здійснюватися лише в межах наперед заданих (т. зв. «стандартних») папок, в яких зберігаються файли заголовків. Зокрема, це папки, у яких зберігаються файли заголовків стандартних бібліотек C/C++. Спосіб запису #include <file> дозволяє під час компіляції програми пришвидшити включення файлів зі стандартних бібліотек C/C++, а також інших зовнішніх бібліотек.

Якщо ж використовується форма запису #include "file", пошук файлу file здійснюється спочатку у поточній папці проєкту (де, як правило, знаходяться .c/.cpp файли), і тільки, якщо потрібний файл у цій папці не знайдено, його пошук буде продовжено в межах наперед заданих «стандартних» папок, в яких зберігаються файли заголовків. Таким чином, форма запису #include "file" дозволяє під час компіляції програми пришвидшити включення файлів, що є складовими поточного проєкту C/C++.

#error

Ця директива зупиняє процес компіляції і виводить на екран повідомлення про помилку. Її синтаксис наступний:

```
#error повідомлення
```

Коли препроцесор зустрічає дану директиву, він зупиняє свою роботу, друкує номер рядка, у якому записана директива, і виводить текст `повідомлення`. Як правило, ця директива застосовується тоді, коли подальша компіляція програми не має сенсу. Приклад використання:

```
#ifndef UNICODE
#error Error! Unicode must be defined!
#endif
```

Якщо символ `UNICODE` є невизначеним, компіляція програми зупиняється і виводиться повідомлення Error! Unicode must be defined!

#line

Директива `#line` задає новий номер рядка і може задавати нове ім'я вихідного файлу. Синтаксис її використання наступний:

```
#line num "filename"
```

Після обробки компілятором вказаного рядка, наступному рядку присвоюється номер `num`, і подальша нумерація рядків продовжується з цього нового номеру. Якщо параметр `"filename"` заданий, він стає новим значенням для внутрішньої змінної `__FILE__` (макрос з ім'ям поточного файлу, що компілюється). Наприклад, директива `#line 120` встановлює номер наступного рядка рівним 120, а директива `#line 120 "Lab06.cpp"` – робить теж саме і ще додатково змінює внутрішнє ім'я файлу (макрос `__FILE__`) на `"Lab06.cpp"`.

#pragma

Директива `#pragma` дозволяє керувати різними аспектами роботи компілятора. Синтаксис використання цієї директиви наступний:

```
#pragma argument
```

Параметр цієї директиви `argument` визначає, що саме і як слід змінити у роботі компілятора. На сьогоднішній день відомо більше 40 допустимих параметрів для директиви `#pragma`. Нижче розглянемо лише деякі з них.

Директива `#pragma` часто використовується, щоб заборонити/дозволити компілятору виводити повідомлення про певні попередження. Дозвіл або заборона виведення таких повідомлень власне не впливатиме на процес компіляції програми (навіть якщо повідомлення виводитись не будуть, програма все одно може не компілюватися). Наприклад, код

```
#pragma warning(disable : 4507 34; error : 164)
```

забороняє компілятору виводити повідомлення про попередження з номерами 4507 та 34, але при цьому попередження з номером 164 буде трактуватися як помилка.

Аргумент `intrinsic` дозволяє використовувати вбудовані версії деяких функцій зі стандартної бібліотеки C, що може покращити швидкодію роботи програми. Такі вбудовані функції включають в себе функції для копіювання блоків пам'яті, деякі функції для роботи з текстовими рядками у форматі C, математичні функції тощо. Наприклад, код

```
#pragma intrinsic(memcpy, cos)
```

вказує компілятору, що замість генерації викликів функцій `memcpy()` та `cos()`, в об'єктний код програми слід безпосередньо вставляти тіло цих функцій.

Останній приклад використання директиви:

```
#pragma once
```

змушує компілятор використовувати вміст файлу заголовку тільки один раз. Тобто якщо деякий файл заголовку із записаною в ньому директивою `#pragma once` включається до декількох .c/.cpp файлів, вміст файлу заголовку під час компіляції буде враховуватись лише один раз, що іноді може суттєво пришвидшити процес компіляції програми.

Наприкінці зазначимо, якщо компілятор зустрічає директиву `#pragma` з невідомим йому аргументом `argument`, він просто ігнорує цю директиву. Отже, використання директив `#pragma` з параметрами, «невідомими» для певної версії компілятора, не призводить до помилки компіляції програми.

Перетворювання в текстовий рядок (#)

Аргументу деякого макроса може передувати символ-оператор `#`, який вказує компілятору на те, що цей аргумент слід перетворити на текстовий рядок (у форматі C). В результаті такої операції замість аргументу компілятор генерує його ім'я у вигляді текстового рядка. Наприклад, код `#Text1` генерує

текстовий рядок "Text1". А макрос

```
#define ShowInt(var) printf( #var " = %d\n", (int)(var))
```

дозволяє перетворити ідентифікатор довільної змінної var на текстовий рядок, у якому записано цей ідентифікатор, і потім вивести значення відповідної змінної за допомогою функції printf(). Наприклад, якщо int i = 1;, оператор ShowInt(i); приведе до виведення на консоль рядка i = 1.

Склеювання лексем (##)

Оператор склеювання або об'єднання лексем ## об'єднує дві лексеми C/C++, між якими він знаходиться, в одну нову лексему. За допомогою цього оператора можна виконувати досить складні перетворення в коді C/C++.

Наприклад, наступний фрагмент коду

```
#define STR(x)          #x
#define STR2(x, y)      STR(x)##STR(y)
```

визначає макрос STR(x) для перетворення деякого об'єкта x у текстовий рядок. Наступний макрос STR2(x, y) об'єднує два текстових рядка, пов'язаних з об'єктами x та y. В результаті виклику функції printf(STR2(18,34)"\n"); на консоль буде виведений текст 1834, який утворюється в результаті об'єднання фрагменту коду (числа) 18 з фрагментом (числом) 34.

Також за допомогою ## можна автоматизувати процедуру створення та використання змінних з певним форматом запису ідентифікаторів. Наприклад, наступний фрагмент коду

```
#define VARi( x ) var_ ## x
int VARi( 1a ) = 1;   // int var_1a = 1;
VARi( 1a ) = 2;       // var_1a = 2;
```

визначає макрос VARi(x), який далі використовується для створення і використання змінної var_1a.

Стандартні макроси C/C++

Мова C/C++ має ряд стандартних макросів, які доступні під час компіляції програми. Якщо в тексті програми препроцесор C/C++ зустрічає один з

таких макросів, він автоматично підставляє замість нього потрібне значення, яке визначається в момент компіляції програми. Ось стислий опис деяких стандартних макросів:

Макрос `__cplusplus` є визначеним, якщо компілятор працює як компілятор C++. Якщо цей макрос не визначений, це означає, що компілятор здійснює компіляцію тексту програми за правилами мови C;

`__DATE__` є текстовим рядком, який містить інформацію про дату обробки файлу з цим макросом препроцесором C/C++;

`__FILE__` є текстовим рядком, який містить інформацію про ім'я файлу, в якому знаходиться цей макрос;

`__LINE__` є цілим числом, в якому зберігається номер поточного рядка у тексті програми;

`__TIME__` є текстовим рядком, який містить інформацію про час, у який препроцесор C/C++ обробляє файл з цим макросом.

ПОБІТОВІ ОПЕРАЦІЇ

Побітові або ***порозрядні операції*** – це операції, які можуть виконуватись над окремими бітами або групами бітів. Деякі такі операції є аналогами логічних операцій (див. лабораторну роботу № 2), які реалізуються для окремих бітів або їх груп. До таких операцій відносяться: побітове заперечення, побітові операції "І" та "АБО". Крім того, в мовах C/C++ визначені додаткові побітові операції: "виключне АБО", побітовий зсув ліворуч та праворуч.

Всі побітові операції виконуються безпосередньо над двійковим представленням величин, які розглядаються як цілі числа без знаку. Враховуючи це, побітові операції рекомендується виконувати над величинами беззнакового (`unsigned`) типу. Для величин іншого (не `unsigned`) типу виконання цих операцій може привести до втрати/зміни формату збереження даних.

Побітове заперечення (побітова інверсія)

Операція побітового заперечення позначається символом тильди `~`. Ця операція `~x` виконує інверсію значень бітів у двійковому представленні величини `x`, тобто змінює наявні значення бітів з 1 на 0 і з 0 на 1. Наприклад, якщо бітове представлення величини `x` є 1011 0111 (x = 183 = 0xB7), операція `~x` дасть результат 0100 1000 (x = 72 = 0x48).

Побітове "І"

Операція побітове "І", яка позначається символом `&`, виконує побітове «множення» двійкових представлень двох величин: `x & y`. Результатом цієї операції для двох бітів (із можливими значеннями 1 або 0) є 1 тільки тоді, коли обидва операнди дорівнюють 1. Інакше результатом виконання операції буде 0.

Наприклад, для фрагментів довжиною 4 біти: `1011 & 1100 == 1000`, `1111 & 1010 == 1010`.

Побітове "АБО"

Операція побітове "АБО", яка позначається символом `|`, виконує побітове «додавання» двійкових представлень двох величин: `x | y`. Результатом цієї операції для двох бітів (із можливими значеннями 1 або 0) є 1, якщо хоча б один з операндів дорівнює 1. Результат виконання цієї операції буде 0, тільки, якщо обидва операнди містять 0.

Наприклад, для фрагментів довжиною 4 біти: `1011 | 1100 == 1111`, `1111 | 1010 == 1111`, `1001 | 1100 == 1101`.

Побітове "виключне АБО"

Операція побітове "виключне АБО", яка позначається символом `^`, виконує побітове «додавання» за модулем 2 двійкових представлень двох величин: `x ^ y`. Результатом цієї операції для двох бітів (із можливими значеннями 1 або 0) є 1, тільки якщо значення обох операндів відрізняються (0 та 1, або 1 та 0). Інакше результат виконання цієї операції буде 0.

Наприклад, для фрагментів довжиною 4 біти: `1011 ^ 1100 == 0111`, `1111 ^ 1010 == 0101`, `1001 ^ 1100 == 0101`.

Важливою властивістю операції "виключне АБО" є її циклічність. Якщо ця операція двічі послідовно застосовується до деякої довільної величини `x`, отриманий результат буде дорівнювати `x`: `(x ^ y) ^ y == x`. Ця властивість "виключного АБО" іноді використовується для шифрування/дешифрування даних.

Побітові зсуви праворуч та ліворуч

Операції побітового зсуву праворуч (`>>`) та ліворуч (`<<`) виконують

зміщення групи бітів на вказану кількість позицій праворуч або ліворуч. При цьому ті біти, які після зміщення потрапляють за межі початкового блоку даних, відкидаються (не враховуються). Якщо ж при зсуві виникають вільні бітові розряди, вони заповнюються нулями.

Наприклад, для фрагментів довжиною 4 біти: `0001 << 1 == 0010`, `0010 << 1 == 0100`, `0010 << 2 == 1000`, `1101 >> 1 == 0110`, `1101 >> 3 == 0001`.

Особливістю операцій побітового зсуву є те, що при зсуві деякої величини ліворуч на n розрядів, ця величина збільшується в 2^n раз. А при зсуві величини праворуч на n розрядів, вона зменшується в 2^n раз. Таким чином, операція `x << 1` збільшує `x` вдвічі, а операція `x >> 2` зменшує `x` в 4 рази.

Бітові маски та їх використання

Бітова маска – це сукупність бітів-«прапорців», яка використовується для виділення, встановлення (в значення 1) або скидання (в 0) певної групи бітів за допомогою побітових операцій. Бітова маска, як правило, являє собою ціле число без знаку, внутрішнє двійкове представлення якого має певний зміст для програміста.

Розглянемо для спрощення деяку величину `x` довжиною 8 біт, наприклад, число 121, яке має внутрішнє двійкове представлення `0111 1001`. Для виділення старшого біта (біта 7) цієї величини введемо маску `y = 1000 0000`, в якій встановлений (у значення 1) лише потрібний біт 7. Використаємо записану маску для виконання різних побітових операцій над `x`. Результати виконання цих операцій зведені в таблицю:

`x`	`0111 1001`	`x`	`0111 1001`
`y`	`1000 0000`	`~y`	`0111 1111`
`x & y`	`0000 0000`	`x & ~y`	`0111 1001`
`x`	`0111 1001`	`x`	`0111 1001`
`y`	`1000 0000`	`~y`	`0111 1111`
`x | y`	`1111 1001`	`x | ~y`	`0111 1111`
`x`	`0111 1001`	`x`	`0111 1001`
`y`	`1000 0000`	`~y`	`0111 1111`
`x ^ y`	`1111 1001`	`x ^ ~y`	`0000 0110`

Як видно, операція `x & y` дає нульовий результат. Це означає, що біт 7 (єдиний встановлений біт у масці `y`) має значення 0 для величини `x`. Операція `x & y` дає можливість перевірити встановлені, чи ні всередині `x` біти, що відповідають масці.

Для встановлення бітів у числі `x` достатньо виконати операцію `x | y` –

тоді ті біти, які були встановлені у масці y, будуть встановлені і в x. Нарешті, операція x & ~y може бути використана для скидання всередині x всіх бітів, встановлених у масці y.

Використовуючи бітові маски та операції побітового зсуву, можна визначити та вивести на консоль двійкове представлення довільного цілого числа без знаку (лістинг 6.1). Відповідний фрагмент коду може бути корисним, тому що в стандартних бібліотеках C/C++ немає вбудованих інструментів, здатних це робити. Цей код для виведення числа в двійковій формі, зокрема, можна використати для вивчення роботи деяких побітових операцій.

Лістинг 6.1:

```
 1: #include <iostream>
 2:
 3: using namespace std;
 4:
 5: void print_bin( unsigned short x )
 6: {
 7:     // mask = bin: 1000 0000 0000 0000
 8:     unsigned short mask = 0x8000;
 9:
10:     for( int i = 1; i <= 16; i++ )
11:     {
12:         // Проаналізувати значення біту.
13:         if( x & mask ) cout << '1';
14:         else cout << '0';
15:         // Зсунути маску.
16:         mask = mask >> 1;
17:         // Якщо потрібно, додати пробіл між
18:         // блоками по 4 біти.
19:         if( (i % 4) == 0 && i != 16) cout << ' ';
20:     }
21: }
22:
23: int main( void )
24: {
25:     unsigned short x, y;
26:
27:     // Вводимо 2 числа без знаку.
28:     cout << "Please input two unsigned short
29: integers:" << endl;
30:     cout << "First short integer  - operand: ";
31:     cin >> x;
32:     cout << "Second short integer - mask    : ";
33:     cin >> y;
```

```
34:
35:     // Виводимо їх в різному форматі.
36:     cout << endl;
37:     cout << "Operand x: " << x << "\t";
38:     print_bin(x);
39:     cout << "   0x" << hex << x << dec << endl;
40:     cout << "Mask y   : " << y << "\t";
41:     print_bin(y);
42:     cout << "   0x" << hex << y << dec << endl;
43:     cout << endl;
44:
45:     // Виконання x & y.
46:     cout << "  x\t";
47:     print_bin(x);
48:     cout << endl;
49:     cout << "  y\t";
50:     print_bin(y);
51:     cout << endl;
52:     cout << "----------------------------" <<
53: endl;
54:     cout << "x & y\t";
55:     print_bin(x & y);
56:     cout << endl << endl;
57:
58:     // Виконання x | y.
59:     cout << "  x\t";
60:     print_bin(x);
61:     cout << endl;
62:     cout << "  y\t";
63:     print_bin(y);
64:     cout << endl;
65:     cout << "----------------------------" <<
66: endl;
67:     cout << "x | y\t";
68:     print_bin(x | y);
69:     cout << endl << endl;
70:
71:     // Виконання x ^ y.
72:     cout << "  x\t";
73:     print_bin(x);
74:     cout << endl;
75:     cout << "  y\t";
76:     print_bin(y);
77:     cout << endl;
78:
```

```
79:       cout << "----------------------------" <<
80:  endl;
81:       cout << "x ^ y\t";
82:       print_bin(x ^ y);
83:       cout << endl << endl;
84:
85:       return 0;
86:  }
```

Найбільш важливим елементом програми з лістингу 6.1 є функція `print_bin()`. Ця функція, використовуючи цикл `for` (рядки 10-20), по черзі перевіряє за допомогою маски (рядки 13-14) чи встановлені певні біти в аргументі `x`. Аналіз починається з самого старшого біту, з яким пов'язана маска `1000 0000 0000 0000`, яка у шістнадцятковій системі числення має вигляд `0x8000`. Якщо результат виконання операції `x & mask` ненульовий, на консоль виводиться символ `'1'`, інакше – символ `'0'`. Після цього маска зсувається праворуч `mask = mask >> 1;`, що дозволяє у подальшому аналізувати стан сусіднього більш молодшого біта в аргументі `x`. Для зручності, після виведення кожних 4 бітових розрядів виводиться додатковий символ пробілу (рядок 19).

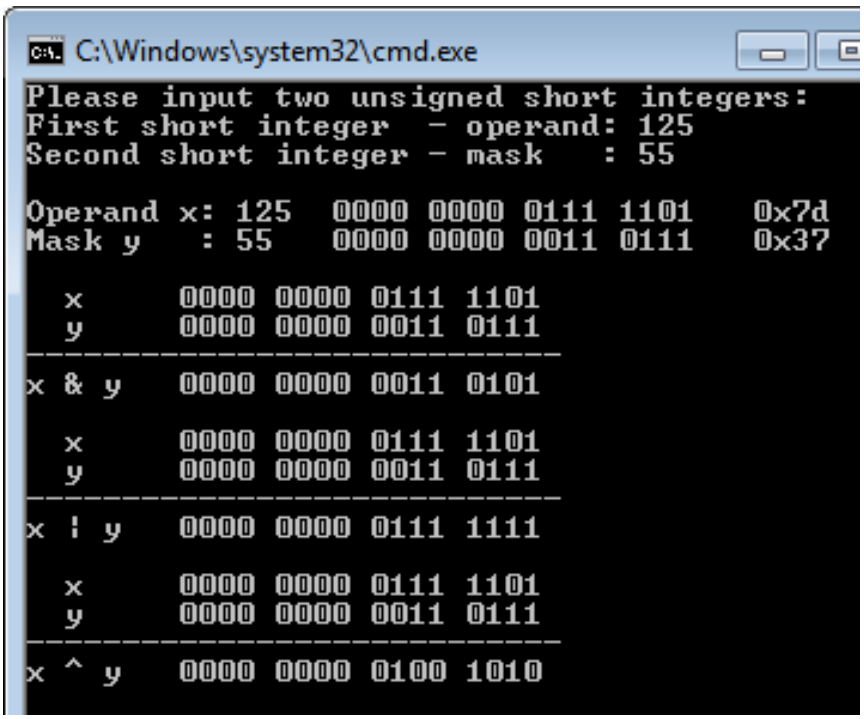


Рис. 24

Результат виконання програми з лістингу 6.1 для двох чисел, 125 (операнд) та 55 (маска), показано на рис. 24.

Наприкінці зазначимо, що для виділення, встановлення або скидання у

деякій величині `unsigned int x` біта з номером `n` (нумерація починається з 0), можуть бути використані наступні макроси:

```
#define CheckBit(x, n)   ((x) & (1UL<<(n)))
#define ClearBit(x, n)   ((x) & ~(1UL<<(n)))
#define SetBit  (x, n)   ((x) | (1UL<<(n)))
```

Використання кольорового тексту та фону при роботі з консоллю

Побітові операції можуть використовуватись для керування станом деякої системи, за умови що цей стан характеризується набором бітових «прапорців». Як приклад такої системи розглянемо системну консоль ОС Windows.

В ОС Windows функція `SetConsoleTextAttribute()` встановлює атрибути кольору тексту на консолі (колір букв і колір фону). Вона має наступний прототип:

```
BOOL SetConsoleTextAttribute(HANDLE, WORD);
```

Функція приймає два аргументи: дескриптор (вікна) консолі – параметр типу `HANDLE`, і число типу `WORD`, біти якого визначають кольори букв і фону. Тип `HANDLE` є аналогом `void*`, а тип `WORD` являє собою 16-бітове ціле число без знаку (`unsigned short`).

Принцип розфарбування тексту на консолі за допомогою `SetConsoleTextAttribute()` наступний: всі символи, виведені в консоль після її виклику, будуть мати встановлені нею атрибути (колір символу та фону).

Для отримання дескриптора системної консолі `hStdOut` можна використовувати функцію `GetStdHandle()`:

```
HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
```

Параметр функції `GetStdHandle()` може приймати значення:
`STD_INPUT_HANDLE` – дескриптор стандартного пристрою вводу даних;
`STD_OUTPUT_HANDLE` – дескриптор стандартного пристрою виводу даних;
`STD_ERROR_HANDLE` – дескриптор стандартної пристрою виводу інформації про помилки.

Другим аргументом у функцію `SetConsoleTextAttribute()` передається число, молодші 8 біт якого визначають колір фону і букв. Щоб пояснити принцип кодування цієї інформації за допомогою 8 біт, представимо цей набір бітів у вигляді таблиці:

Номер біта	7	6	5	4	3	2	1	0
Позначення	I_b	R_b	G_b	B_b	I_f	R_f	G_f	B_f
Елемент тексту	Фон				Символи			

Індекс *b* у записаних позначеннях бітів означає, що такий біт впливає на властивості фону (background). Індекс *f* (від foreground) визначає властивості символів, які виводяться поверх фону.

Інтенсивність (яскравість) фону та символу задаються бітами I_b та I_f (1 – яскравий, 0 – ні). Наявність червоної (red), зеленої (green), синьої (blue) складових кольору фону визначаються бітами R_b, G_b, B_b, а кольору символів, відповідно, бітами R_f, G_f, B_f. Таким чином, яскравість та кольори фону й символів можна задати за допомогою 8-бітового числа без знаку, яке приймає значення від 0 до 255. Наприклад, якщо функції `SetConsoleTextAttribute()` другим параметром передати число 2, яке в двійковому представлені виглядає як 0000 0010, символи будуть виводитись темно-зеленим кольором (G_f = 1, R_f = 0, B_f = 0, біт яскравості I_f = 0) на чорному фоні (R_b = G_b = B_b = I_f = 0).

Іншим способом змінити колір символів та фону при виведенні даних на консоль, є використання стандартної функції C/C++ `system()`, оголошеної у файлі `stdlib.h`. Функція `system(char* str)` передає рядок `str` командному процесору операційної системи (в ОС Windows це cmd.exe). Якщо `system()` викликається з аргументом `NULL`, вона повертає ненульове значення, коли командний процесор доступний, і нуль, коли він не доступний.

Будь які операції, які можуть бути виконані за допомогою командного процесора, можна виконати й функцією `system()`. Наприклад, за допомогою `system()` можна виводити кольоровий текст на кольоровому фоні з використанням таких кольорів:

Код	**Колір**	**Код**	**Колір**
0	Чорний	8	Сірий
1	Синій	9	Світло-синій
2	Зелений	A	Світло-зелений
3	Блакитний	B	Світло-блакитний
4	Червоний	C	Світло-червоний
5	Фіолетовий	D	Світло-фіолетовий
6	Жовтий	E	Світло-жовтий
7	Білий	F	Яскравий білий

Наприклад, виклик `system("Color 1E")` задає виведення символів жовтого кольору на синьому фоні (перший код 1 відповідає фону, другий – E – тексту). Стандартним форматом вважається такий, що задається як `system("Color 07")`, тобто як білий текст на чорному фоні.

РОБОТА З ФАЙЛАМИ

Файл – це іменований блок інформації, який зберігається на носії інформації. Кожен файл має фіксоване ім'я – послідовність символів, що однозначно характеризує файл. У файлах можуть зберігатися будь-які дані: коди програм, тексти документів, закодовані зображення, аудіо- та відео-записи тощо. В кінці файлу знаходиться спеціальний код, який у мові С позначається за допомогою константи `EOF` (End Of File – англ. кінець файлу).

Існує декілька підходів для роботи з файлами, які ґрунтуються на використанні різних програмних пакетів і бібліотек. В даній лабораторній роботі розглянемо роботу з файлами за допомогою функцій стандартної бібліотеки С. Пізніше, в одній з наступних лабораторних робіт, покажемо, як працювати з файлами використовуючи потоки введення/виведення С++ зі стандартної бібліотеки С++.

У стандартній бібліотеці С основні константи, типи даних та функції для роботи з файлами наведені в файлі заголовку `stdio.h`.

Для роботи з файлами застосовується спеціальний тип даних `FILE`. У програмі мовою С/С++ з файлом пов'язується «файлова змінна», яка має тип `FILE*` і є вказівником на об'єкт типу `FILE`. Виконання будь-яких операцій з файлом здійснюється через цю «файлову змінну». Як правило, для цього «файлова змінна» та інші потрібні параметри передаються деякій функції зі стандартної бібліотеки С.

Відкриття та закриття файлів

Для використання в програмі на С/С++ файлів, необхідно зв'язати «файлову змінну» з тим, чи іншим файлом на носії інформації. Для цього використовується функція `fopen()`, яка має два параметри у вигляді текстових рядків у форматі С, `"file_name"` та `"mode"`:

```
FILE* f = fopen("file_name", "mode");
```

Параметр `"file_name"` має бути дозволеним іменем файлу на носії інформації. Це ім'я може бути як коротким, наприклад, `"my file.txt"`, так і включати в себе повний шлях до файлу, наприклад, `"d:\\My Projects\\Lab06\\my file.txt"`. Звернемо увагу, що елементи шляху до файлу мають розділятись подвійним символом зворотного слеша `\\`. Також можна використовувати альтернативний формат запис шляху за допомогою одинарного прямого слеша `/`, наприклад, `"d:/My Projects/Lab06/my file.txt"`.

Другий параметр `"mode"` містить набір символів, які задають режим

обробки файлу. Зокрема, `"r"` означає відкриття існуючого файлу для читання; `"w"` – перезапис існуючого файлу або відкриття нового файлу; `"a"` – відкриття існуючого файлу для доповнення або створення нового файлу. Додаткові символи `"t"` і `"b"` використовуються для роботи з текстовими чи бінарними (двійковими) файлами. Додатковий символ `"+"` означає, що дані у файл можуть як записуватись, так і зчитуватись з нього. Якщо файл з вказаним ім'ям відсутній у системі, а параметр `"mode"` містить символи `"w"` або `"a"`, буде створено пустий файл, відкритий для запису.

Якщо виклик функції `fopen()` успішний, вона повертає ненульовий вказівник на об'єкт типу `FILE`. Якщо ж при роботі `fopen()` виникла помилка, вона повертає нульовий вказівник (`NULL`). Враховуючи це, необхідно завжди перевіряти те значення, яке повертає `fopen()`.

Ось приклад коду

```
FILE* f1 = fopen("temp.txt", "rt");
FILE* f2 = fopen("c:/Prog/file.dat", "wb");
```

в якому створюються дві «файлові змінні» `f1` та `f2`. Перша змінна `f1` пов'язується з існуючим текстовим файлом temp.txt, який відкривається лише для зчитування з нього даних (`"rt"`). Друга змінна `f2` вказує на файл `"c:/Prog/file.dat"`, який планується перезаписувати як бінарний файл даних (`"wb"`).

Наприкінці роботи з файлом його завжди необхідно закрити, щоб гарантувати збереження записаної у нього інформації. Для закриття файлу і звільнення усіх пов'язаних з ним ресурсів, використовується функція `fclose()`, єдиним аргументом якої є раніше створена «файлова змінна». Наприклад, фрагмент коду

```
fclose(f);
```

закриває файл і звільняє пам'ять, пов'язану з деякою файловою змінною `f`. При вдалому закритті файлу функція `fclose()` повертає 0, при виникненні помилки – константу `EOF`. Ці дані можна використовувати для перевірки успішності закриття файлу.

Введення/виведення даних у файл

Дані, з якими працює програма, зберігаються в файлі у форматі, який передбачено типом файлу. Зокрема, всі дані можна розділити на *двійкові* (*бінарні*), які зберігаються у вигляді набору байтів, та *текстові*, які зберігаються у вигляді послідовності символів. Відповідно, є *бінарні* та *текстові файли*.

Останні виділяють як окремий тип файлів через значну кількість практичних задач, пов'язаних з обробкою текстів.

При роботі з бінарними файлами дані можна зчитувати та записувати безпосередньо, не виконуючи їх форматування. Відповідно, цей спосіб зчитування та збереження даних є доволі швидким. При роботі з текстовими файлами треба здійснювати перетворення даних з символьного вигляду в один із стандартних форматів даних C/C++, або навпаки. Завдяки цьому обробка текстових файлів виконується повільніше, ніж бінарних.

Зчитування бінарних даних. Для зчитування з файлу блоку бінарних даних використовується функція `fread()`. Її параметрами є адреса буфера, куди будуть збережені дані, що зчитуються; кількість байт даних, які потрібно зчитати (залежить від типу даних); кількість елементів для зчитування та «файлова змінна». Функція `fread()` повертає число, яке дорівнює кількості реально зчитаних з файлу елементів (це число може відрізнятися від заданого, так як файл може виявитися коротшим, ніж очікувалося). Якщо під час зчитування виникла помилка, `fread()` може повертати значення нуль. Для перевірки коректності виклику `fread()` може також використовуватись функція `ferror()`, єдиним параметром якої є «файлова змінна».

Наведений нижче фрагмент коду демонструє використання `fread()` для зчитування даних з файлу, асоційованого з «файловою змінною» `f`. Спочатку дані (ціле число) зчитуються у змінну `int i`. Другий виклик `fread()` приводить до зчитування з файлу `512` значень типу `float` в масив `float arr[512]`:

```
unsigned int number;
int i;
float arr[512];
number = fread(&i, sizeof(i), 1, f);
number = fread(arr, sizeof(float), 512, f);
```

Запис бінарних даних у файл здійснюється за допомогою функції `fwrite()`. Вона має 4 аргументи, порядок та зміст яких аналогічні до аргументів функції `fread()`. Тільки тепер буфер, адреса якого передається першим параметром у `fwrite()`, має містити дані, які слід записати у файл.

Функція `fwrite()` повертає число, яке дорівнює кількості реально записаних у файл елементів. Якщо під час запису даних виникла помилка, `fwrite()` може повертати значення нуль. Для перевірки коректності виклику `fwrite()` може також використовуватись функція `ferror()`, єдиним параметром якої є «файлова змінна».

Наведений нижче фрагмент коду демонструє використання `fwrite()`:

```
unsigned int number;
int i;
```

```
float arr[512];
number = fwrite(&i, sizeof(i), 1, f);
number = fwrite(arr, sizeof(float), 512, f);
```

Спочатку у файл записується вміст змінної `int i`, а потім – вміст масиву `float arr[512]`.

Зчитування даних з текстових файлів відбувається за допомогою функції форматного введення даних `fscanf()`. Ця функція перетворює дані, зчитані з текстового файлу, у вказаний формат. Використання функції `fscanf()` аналогічно до використання функції зчитування даних з консолі `scanf()`. Першим параметром `fscanf()` має задаватись «файлова змінна»; інші її параметри такі ж самі як для `scanf()`. Нижче наводиться приклад використання `fscanf()` для зчитування значення цілого числа у змінну `int i`:

```
FILE* f;
int i;
fscanf(f, "%d", &i);
```

Для зчитування з текстового файлу суто символьної інформації також використовуються функції `fgetc()`, `fgets()` або їх більш безпечні аналоги на кшталт `fgets_s()`. Наприклад, фрагмент коду

```
char c = fgetc(f);
```

виконує зчитування з файлу, асоційованого з «файловою змінною» `f`, один символ у змінну `c` типу `char`. А фрагмент коду

```
char str[100];
fgets(str, 100, f);
```

реалізує зчитування через «файлову змінну» `f` до 99 символів у масив `char str[100]`. Після цього `fgets()` автоматично ініціалізує останній елемент масиву нульовим символом `'\0'`, щоб отримати текстовий рядок у форматі C. Якщо текстовий рядок, записаний у файлі, містить менше символів, ніж вказаний другим параметром розмір буфера, `fgets()` зчитає в буфер тільки дані текстового рядка з файлу, включаючи і символ кінця рядка `'\0'`.

Запис даних у текстовий файл можна здійснити за допомогою функції `fprintf()`. Вона перетворює вхідні дані в текстове (символьне) представлення, відповідно до вказаного формату. Принцип роботи цієї функції в цілому аналогічний до принципу роботи функції `printf()`, однак, `fprintf()` виводить текстове повідомлення не на консоль, а у файл. Першим параметром у `fprintf()` передається «файлова змінна», а всі інші її параметри – такі ж самі як і у `printf()`. Ось приклад використання `fprintf()` для запису у

файл, пов'язаний з «файловою змінною» `f`, значення змінних `int i` та `double d`:

```
int    i = 10;
double d = 3.14;
fprintf(f, "%d %g", i, d);
```

Запис у текстовий файл символьної інформації також може здійснюватися функціями `fputc(c, f)` та `fputs(str, f)`. Перша дозволяє записати у файл, пов'язаний з «файловою змінною» `f`, символ `c`, а інша – записати у файл текстовий рядок `str` в форматі С.

Робота з маркером файлу

Окрім інформації про ім'я файлу та режим доступу до файлу, «файлова змінна» також містить інформацію про поточне місце у файлі (номер байту), де в даний момент може виконуватись зчитування або запис даних. Це положення асоціюється з т. зв. *маркером файлу* або просто *маркером*.

Нумерація байтів, а отже й записів, у файлі починається з нуля. При відкритті файлу чи створенні нового файлу, маркер зазвичай встановлюється на нульовий запис – на початок файлу. При відкритті існуючого файлу для доповнення, маркер встановлюється у кінець файлу. При виконанні операцій зчитування або запису даних маркер файлу автоматично зміщується вперед на ту кількість байт, які були зчитані з файлу або записані у файл.

Як вже зазначалось, при зчитуванні/записі заданої кількості байт маркер зміщується автоматично. Але інколи виникає необхідність зчитувати або записувати дані у файл непослідовно. У таких випадках зручною є функція `fseek()`, яка виконує примусове позиціонування маркера в потрібне місце всередині файлу. Першим аргументом цієї функції є «файлова змінна», другим – величина зміщення маркера типу `signed int` (тобто ціле число зі знаком), а третій параметр характеризує позицію всередині файлу, від якої буде здійснюватися зміщення маркера. Для останнього параметра передбачено три можливих значення: константа `SEEK_SET` означає відлік позиції маркера від початку файлу; `SEEK_CUR` означає зсув позиції відносно поточного положення маркера, а `SEEK_END` означає відлік зміщення відносно кінця файлу (в цьому випадку це зміщення має бути від'ємним). Так, фрагмент коду

```
fseek(f, 5, SEEK_CUR);
```

зміщує маркер, пов'язаний із «файловою змінною» `f`, на 5 байт вперед відносно поточної позиції. Виклик функції

```
fseek(f, n*sizeof(int), SEEK_SET);
```

встановлює маркер файлу на початок запису з номером `n`, де кожен запис має розмір `sizeof(int)`. Нульовому запису (початку файлу) відповідає нульове значення `n`. Нарешті, фрагмент коду

```
fseek(f, 0, SEEK_SET);
```

встановлює маркер на початок файлу, а фрагмент коду

```
fseek(f, 0, SEEK_END);
```

– на кінець файлу.

Щоб дізнатися, чи знаходиться маркер в кінці файлу, можна скористатись функцією `feof(f)`, яка повертає значення істини, якщо маркер стоїть в кінці файлу.

Програма у лістингу 6.2 демонструє приклад роботи з текстовими файлами. Ця програма виконує шифрування/дешифрування текстових файлів.

Лістинг 6.2:

```
 1: #define _CRT_SECURE_NO_WARNINGS
 2: #include <stdio.h>
 3:
 4: void CryptTextFile( const char* fname1, const
 5: char* fname2 )
 6: {
 7:     char c, key;
 8:
 9:     // Відкрити початковий файл.
10:     FILE* f1 = fopen(fname1, "rt");
11:     if( !f1 )
12:     {
13:         printf("Error opening file.");
14:         return;
15:     }
16:
17:     // Створити файл з шифруванням.
18:     FILE* f2 = fopen(fname2, "wt+");
19:     if( !f2 )
20:     {
21:         printf("Error creating file.");
22:         fclose(f1);
23:         return;
24:     }
```

```
25:
26:       key = 0;
27:       while( !feof(f1) )// Поки не зустрінемо EOF.
28:       {
29:
30:           // Зчитати символ.
31:           c = (char)fgetc(f1);
32:           if( feof(f1) ) break;
33:           // Зашифрувати його.
34:           c = c ^ key;
35:           // Записати зашифрований символ у файл.
36:           fputc((int)c, f2);
37:           // Змінити ключ.
38:           if( key == 255 ) key = 0;
39:           else key++;
40:       }
41:
42:       // Закрити файли.
43:       if( f1 ) fclose(f1);
44:       f1 = NULL;
45:       if( f2 ) fclose(f2);
46:       f2 = NULL;
47:  }
48:
49:
50:  int main( void )
51:  {
52:       // Зашифрувати файл.
53:       CryptTextFile("test.txt", "crypt.txt");
54:       // Повторити процедуру.
55:       CryptTextFile("crypt.txt", "test2.txt");
56:
57:       return 0;
58:  }
```

Функція `main()` цієї програми реалізує лише два виклика функції `CryptTextFile()`, яка і виконує всю основну роботу. Під час першого її виклику (рядок 53), дані з текстового файлу test.txt шифруються і зберігаються у текстовому файлі crypt.txt. Наступний виклик функції `CryptTextFile()` приводить до дешифрування вмісту файлу crypt.txt і збереження дешифрованого фрагменту тексту у файлі test2.txt.

Шифрування/дешифрування даних відбуваються за рахунок виконання операції "виключне АБО" над символом, зчитаним з текстового файлу, та ключем `key`. Оскільки операція "виключне АБО" має властивість циклічності, що

пояснюється тотожністю `x == (x ^ y) ^ y` для довільних `x` та `y`, повторний виклик функції `CryptTextFile()` для файлу із зашифрованими даними, приводить до дешифрування його вмісту. В результаті вміст файлів test.txt та test2.txt має співпадати.

В функції `CryptTextFile()` спочатку відкриваються обидва файли: вхідний з іменем, що задається параметром `fname1`, та кінцевий, з іменем `fname2`, в якому мають зберігатись зашифровані дані. Відкриття файлів відбувається за допомогою виклику функції `fopen()` (рядки 10, 18). Після виклику `fopen()` проводиться перевірка «файлових змінних» `f1` та `f2`.

Далі, в циклі `while`, поки не досягнуто кінець вхідного файлу, зчитуються символи за допомогою виклику `fgetc()` у рядку 31. Якщо зчитаний символ не є останнім у файлі, цей символ шифрується за допомогою операції "виключне АБО": `c = c ^ key;` і потім записується у кінцевий файл (рядок 36). Наприкінці циклу `while` відбувається циклічна зміна ключа шифрування `key` (рядки 38-39).

Перед завершенням своєї роботи функція `CryptTextFile()` закриває обидва файли (рядки 43, 45), використовуючи функцію `fclose()` зі стандартної бібліотеки C.

Завдання

Варіант 1

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як скалярний добуток, розрахунок модуля вектора, множення вектора на число, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (5, –3, 1) та (8, 8, 3) виконати скалярне та векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 3.
3) Результати трьох операцій над векторами вивести на червоному, чорному та синьому фоні консолі зеленим кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-1.dat і текстовий файл Lab06-1.txt.
5) По черзі зчитати дані з файлів Lab06-1.dat та Lab06-1.txt і відобразити їх на екрані чорним шрифтом на білому фоні.

Варіант 2

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як розрахунок модуля вектора, множення вектора на число, знаходження кута між векторами, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (1, –2, 3) та (4, 3, –5) виконати векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 2. Розрахувати кут між вказаними векторами.
3) Результати трьох операцій над векторами вивести на зеленому, жовтому та білому фоні консолі червоним кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-2.dat і текстовий файл Lab06-2.txt.
5) По черзі зчитати дані з файлів Lab06-2.dat та Lab06-2.txt і відобразити їх на екрані яскраво білим шрифтом на чорному фоні.

Варіант 3

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як скалярний добуток, розрахунок модуля вектора, множення вектора на число, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (–3, 7, 4) та (–2, 6, –5) виконати скалярне та векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 4.
3) Результати трьох операцій над векторами вивести на білому, жовтому та червоному фоні консолі синім кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-3.dat і текстовий файл Lab06-3.txt.
5) По черзі зчитати дані з файлів Lab06-3.dat та Lab06-3.txt і відобразити їх на екрані жовтим шрифтом на чорному фоні.

Варіант 4

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як розрахунок модуля вектора, множення вектора на число, знаходження кута між векторами, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (9, 8, 7) та (–7, –8, –9) виконати векторне множення, знайти модулі векторів, а також здійснити множення векторів на

число 5. Розрахувати кут між вказаними векторами.

3) Результати трьох операцій над векторами вивести на блакитному, пурпуровому та зеленому фоні консолі білим кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2 записати в бінарний файл Lab06-4.dat і текстовий файл Lab06-4.txt.
5) По черзі зчитати дані з файлів Lab06-4.dat та Lab06-4.txt і відобразити їх на екрані чорним шрифтом на сірому фоні.

Варіант 5

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як скалярний добуток, розрахунок модуля вектора, множення вектора на число, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (2, 0, 1) та (5, –6, –3) виконати скалярне та векторне множення, знайти модулі векторів, а також здійснити множення векторів на число –1.
3) Результати трьох операцій над векторами вивести на зеленому, білому та червоному фоні консолі чорним кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-5.dat і текстовий файл Lab06-5.txt.
5) По черзі зчитати дані з файлів Lab06-5.dat та Lab06-5.txt і відобразити їх на екрані білим шрифтом на синьому фоні.

Варіант 6

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як розрахунок модуля вектора, множення вектора на число, знаходження кута між векторами, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (1, 5, –7) та (–4, 8, 1) виконати векторне множення, знайти модулі векторів, а також здійснити множення векторів на число –2. Розрахувати кут між вказаними векторами.
3) Результати трьох операцій над векторами вивести на жовтому фоні консолі червоним, чорним та синім кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-6.dat і текстовий файл Lab06-6.txt.
5) По черзі зчитати дані з файлів Lab06-6.dat та Lab06-6.txt і відобразити їх на екрані жовтим шрифтом на синьому фоні.

Варіант 7

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як скалярний добуток, розрахунок модуля вектора, множення вектора на число, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (2, 6, 9) та (–3, 4, 5) виконати скалярне та векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 6.
3) Результати трьох операцій над векторами вивести на блакитному фоні консолі червоним, чорним та зеленим кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-7.dat і текстовий файл Lab06-7.txt.
5) По черзі зчитати дані з файлів Lab06-7.dat та Lab06-7.txt і відобразити їх на екрані синім шрифтом на жовтому фоні.

Варіант 8

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як розрахунок модуля вектора, множення вектора на число, знаходження кута між векторами, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (4, –5, 8) та (–6, 3, –9) виконати векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 7. Розрахувати кут між вказаними векторами.
3) Результати трьох операцій над векторами вивести на фіолетовому фоні консолі білим, жовтим та червоним кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-8.dat і текстовий файл Lab06-8.txt.
5) По черзі зчитати дані з файлів Lab06-8.dat та Lab06-8.txt і відобразити їх на екрані світло-фіолетовим шрифтом на чорному фоні.

Варіант 9

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як скалярний добуток, розрахунок модуля вектора, множення вектора на число, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (–3, 2, –3) та (1, 9, –2) виконати скалярне та векторне множення, знайти модулі векторів, а також здійснити множення

векторів на число 5.

3) Результати трьох операцій над векторами вивести на білому фоні консолі блакитним, фіолетовим та зеленим кольором тексту. Також вивести результати перевірки того, перевищують чи не перевищують модулі отриманих векторів задане число, із відповідним коментарем. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-9.dat і текстовий файл Lab06-9.txt.
5) По черзі зчитати дані з файлів Lab06-9.dat та Lab06-9.txt і відобразити їх на екрані світло-жовтим шрифтом на синьому фоні.

Варіант 10

1) Використовуючи директиви препроцесора створити макроси, за допомогою яких визначити основні математичні операції над тривимірними векторами, такі як розрахунок модуля вектора, множення вектора на число, знаходження кута між векторами, векторний добуток. Вектори задаються трьома значеннями своїх декартових проекцій.
2) Для векторів з проекціями (4, 0, 6) та (–3, 7, –5) виконати векторне множення, знайти модулі векторів, а також здійснити множення векторів на число 9. Розрахувати кут між вказаними векторами.
3) Результати трьох операцій над векторами вивести на зеленому фоні консолі червоним, чорним та синім кольором тексту. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Результати, отримані в п. 2, записати в бінарний файл Lab06-10.dat і текстовий файл Lab06-10.txt.
5) По черзі зчитати дані з файлів Lab06-9.dat та Lab06-9.txt і відобразити їх на екрані блакитним шрифтом на чорному фоні.

Варіант 11

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти обидва корені квадратного рівняння виду $Ax^2 + Bx + C = 0$ із заданими сталими *A, B* та *C*. Якщо дискримінант від'ємний, вивести у консоль відповідне повідомлення.
2) Створити макроси, які знаходять положення мінімуму функції $Ax^2 + Bx + C$, а також значення функції в мінімумі.
3) Для заданого набору параметрів $A = 4$, $B = 10$, $C = 6$, обчислити корені рівняння та вивести результат на консоль синім кольором тексту на зеленому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Визначити точку мінімуму вказаної функції та вивести результат на консоль червоним кольором тексту на синьому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.

5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-11.dat і текстовий файл Lab06-11.txt.
6) По черзі зчитати дані з файлів Lab06-11.dat та Lab06-11.txt і відобразити їх на екрані білим шрифтом на синьому фоні.

Варіант 12

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти обидва корені квадратного рівняння виду $Ax^2 + Bx + C = 0$ із заданими сталими *A, B* та *C*. Якщо дискримінант від'ємний, вивести на консоль відповідне повідомлення.
2) Створити макроси, які знаходять положення мінімуму функції $Ax^2 + Bx + C$, а також значення функції в мінімумі.
3) Для заданого набору параметрів *A* = 2, *B* = 8, *C* = 3, обчислити корені рівняння та вивести результат на консоль чорним кольором тексту на жовтому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Визначити точку мінімуму вказаної функції та вивести результат на консоль зеленим кольором тексту на білому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-12.dat і текстовий файл Lab06-12.txt.
6) По черзі зчитати дані з файлів Lab06-12.dat та Lab06-12.txt і відобразити їх на екрані червоним шрифтом на чорному фоні.

Варіант 13

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти обидва корені квадратного рівняння виду $Ax^2 + Bx + C = 0$ із заданими сталими *A, B* та *C*. Якщо дискримінант від'ємний, вивести на консоль відповідне повідомлення.
2) Створити макроси, які знаходять положення мінімуму функції $Ax^2 + Bx + C$, а також значення функції в мінімумі.
3) Для заданого набору параметрів *A* = 1, *B* = 5, *C* = 3, обчислити корені рівняння та вивести результат на консоль зеленим кольором тексту на синьому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Визначити точку мінімуму вказаної функції та вивести результат на консоль синім кольором тексту на червоному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-13.dat і текстовий файл Lab06-13.txt.
6) По черзі зчитати дані з файлів Lab06-13.dat та Lab06-13.txt і відобразити

їх на екрані зеленим шрифтом на чорному фоні.

Варіант 14

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти обидва корені квадратного рівняння виду $Ax^2 + Bx + C = 0$ із заданими сталими *A, B* та *C*. Якщо дискримінант від'ємний, вивести на консоль відповідне повідомлення.
2) Створити макроси, які знаходять положення мінімуму функції $Ax^2 + Bx + C$, а також значення функції в мінімумі.
3) Для заданого набору параметрів *A* = 5, *B* = 8, *C* = 2, обчислити корені рівняння та вивести результат на консоль жовтим кольором тексту на чорному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Визначити точку мінімуму вказаної функції та вивести результат на консоль білим кольором тексту на зеленому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-14.dat і текстовий файл Lab06-14.txt.
6) По черзі зчитати дані з файлів Lab06-14.dat та Lab06-14.txt і відобразити їх на екрані чорним шрифтом на зеленому фоні.

Варіант 15

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти обидва корені квадратного рівняння виду $Ax^2 + Bx + C = 0$ із заданими сталими *A, B* та *C*. Якщо дискримінант від'ємний, вивести на консоль відповідне повідомлення.
2) Створити макроси, які знаходять положення мінімуму функції $Ax^2 + Bx + C$, а також значення функції в мінімумі.
3) Для заданого набору параметрів *A* = 3, *B* = 9, *C* = 5, обчислити корені рівняння та вивести результат на консоль чорним кольором тексту на білому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Визначити точку мінімуму вказаної функції та вивести результат на консоль жовтим кольором тексту на червоному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-15.dat і текстовий файл Lab06-15.txt.
6) По черзі зчитати дані з файлів Lab06-15.dat та Lab06-15.txt і відобразити їх на екрані блакитним шрифтом на жовтому фоні.

Варіант 16

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти суму, різницю, добуток та частку двох комплексних чисел $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$, де x_1, x_2, y_1, y_2 – дійсні числа, а i – уявна одиниця. Комплексні числа задаються їх дійсною та уявною частиною, тобто двома величинами x та y.
2) Створити макрос, який розраховує модуль комплексного числа $|z| = |x + iy| = \sqrt{x^2 + y^2}$.
3) Для двох заданих комплексних чисел $z_1 = 2 + i3$ та $z_2 = 4 + i5$ обчислити модулі та вивести результат на консоль чорним кольором тексту на білому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Для тих же самих чисел знайти $z_1 + z_2$, $z_1 - z_2$, $z_1 \times z_2$ та z_1 / z_2 і вивести результат на консоль жовтим кольором тексту на червоному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-16.dat і текстовий файл Lab06-16.txt.
6) По черзі зчитати дані з файлів Lab06-16.dat та Lab06-16.txt і відобразити їх на екрані чорним шрифтом на блакитному фоні.

Варіант 17

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти суму, різницю, добуток та частку двох комплексних чисел $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$, де x_1, x_2, y_1, y_2 – дійсні числа, а i – уявна одиниця. Комплексні числа задаються їх дійсною та уявною частиною, тобто двома величинами x та y.
2) Створити макрос, який розраховує модуль комплексного числа $|z| = |x + iy| = \sqrt{x^2 + y^2}$.
3) Для двох заданих комплексних чисел $z_1 = 6 + i1$ та $z_2 = 2 + i7$ обчислити модулі та вивести результат на консоль зеленим кольором тексту на жовтому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Для тих же самих чисел знайти $z_1 + z_2$, $z_1 - z_2$, $z_1 \times z_2$ та z_1 / z_2 і вивести результат на консоль білим кольором тексту на синьому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-17.dat і текстовий файл Lab06-17.txt.
6) По черзі зчитати дані з файлів Lab06-17.dat та Lab06-17.txt і відобразити їх на екрані синім шрифтом на жовтому фоні.

Варіант 18

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти суму, різницю, добуток та частку двох комплексних чисел $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$, де x_1, x_2, y_1, y_2 – дійсні числа, а i – уявна одиниця. Комплексні числа задаються їх дійсною та уявною частиною, тобто двома величинами x та y.
2) Створити макрос, який розраховує модуль комплексного числа $|z| = |x + iy| = \sqrt{x^2 + y^2}$.
3) Для двох заданих комплексних чисел $z_1 = 4 + i2$ та $z_2 = 1 + i3$ обчислити модулі та вивести результат на консоль жовтим кольором тексту на зеленому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Для тих же самих чисел знайти $z_1 + z_2$, $z_1 - z_2$, $z_1 \times z_2$ та z_1 / z_2 і вивести результат на консоль синім кольором тексту на білому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-18.dat і текстовий файл Lab06-18.txt.
6) По черзі зчитати дані з файлів Lab06-18.dat та Lab06-18.txt і відобразити їх на екрані синім шрифтом на білому фоні.

Варіант 19

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти суму, різницю, добуток та частку двох комплексних чисел $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$, де x_1, x_2, y_1, y_2 – дійсні числа, а i – уявна одиниця. Комплексні числа задаються їх дійсною та уявною частиною, тобто двома величинами x та y.
2) Створити макрос, який розраховує модуль комплексного числа $|z| = |x + iy| = \sqrt{x^2 + y^2}$.
3) Для двох заданих комплексних чисел $z_1 = 8 + i3$ та $z_2 = 5 + i4$ обчислити модулі та вивести результат на консоль чорним кольором тексту на жовтому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Для тих же самих чисел знайти $z_1 + z_2$, $z_1 - z_2$, $z_1 \times z_2$ та z_1 / z_2 і вивести результат на консоль синім кольором тексту на червоному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-19.dat і текстовий файл Lab06-19.txt.
6) По черзі зчитати дані з файлів Lab06-19.dat та Lab06-19.txt і відобразити їх на екрані білим шрифтом на синьому фоні.

Варіант 20

1) Використовуючи директиви препроцесора, створити макроси, за допомогою яких можна знайти суму, різницю, добуток та частку двох комплексних чисел $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$, де x_1, x_2, y_1, y_2 – дійсні числа, а i – уявна одиниця. Комплексні числа задаються їх дійсною та уявною частиною, тобто двома величинами x та y.
2) Створити макрос, який розраховує модуль комплексного числа $|z| = |x + iy| = \sqrt{x^2 + y^2}$.
3) Для двох заданих комплексних чисел $z_1 = 6 + i2$ та $z_2 = 2 + i6$ обчислити модулі та вивести результат на консоль жовтим кольором тексту на червоному фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
4) Для тих же самих чисел знайти $z_1 + z_2$, $z_1 - z_2$, $z_1 \times z_2$ та z_1 / z_2 і вивести результат на консоль червоним кольором тексту на синьому фоні. Задання кольору символів та фону реалізувати за допомогою побітових операцій.
5) Результати, отримані в п. 3-4, записати в бінарний файл Lab06-20.dat і текстовий файл Lab06-20.txt.
6) По черзі зчитати дані з файлів Lab06-20.dat та Lab06-20.txt і відобразити їх на екрані червоним шрифтом на чорному фоні.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке директиви препроцесора і для чого вони потрібні? Наведіть приклади їх використання.
2. Поясніть як за допомогою директиви `#define` можна створювати символьні константи та макроси, що мають один або кілька аргументів.
3. В чому схожість та відмінність у застосуванні макросів та звичайних функцій? Поясніть чи завжди використання макросів є безпечним?
4. Що таке умовна компіляція? Як реалізувати умовну компіляцію певного фрагменту коду програми за допомогою директив препроцесора?
5. Що таке бітові маски і для чого вони використовуються?
6. Поясніть яким чином можна встановлювати, виділяти та скидати значення певних бітів за допомогою побітових операцій.
7. Поясніть в чому відмінність та схожість між операціями `~` та `!`, `&` та `&&`, `|` та `||`.
8. Яким чином можна змінити кольори тексту та фону в консольному застосунку?
9. Поясніть принцип роботи з файлами, використовуючи функції стандартної бібліотеки C.
10. Що таке «файлова змінна» типу `FILE*`? Для чого вона використовується?

Наведіть приклади.

11. Які режими доступу до файлу можуть задаватися при виклику функції `fopen()`? Поясніть чим принципово відрізняються ці режими доступу.
12. У чому полягає основна відмінність між текстовими і бінарними файлами? Чим відрізняються текстові та бінарні файли з точки зору роботи з ними за допомогою функцій стандартної бібліотеки С?
13. Що таке маркер файлу? Поясніть для чого він використовується. Наведіть приклади того, як за його допомогою можна непослідовно зчитувати або записувати дані у файл.
14. Що відбудеться якщо спробувати відкрити неіснуючий файл для читання? Що відбудеться, якщо спробувати відкрити неіснуючий файл для запису?
15. Навіщо після завершення роботи з файлом його потрібно закривати?

Лабораторна робота № 7.
Структуровані типи даних мови C/C++: переліки, структури, об'єднання та бітові поля

Мета роботи:

1) Навчитись перейменовувати типи за допомогою `typedef`.
2) Ознайомитись з поняттям переліків або зчислених типів (`enum`), структур (`struct`), об'єднань (`union`) та бітових полів у мові C/C++.
3) Навчитись створювати нові структуровані типи даних та використовувати їх у власних програмах на C/C++.

Структуровані та нестандартні типи даних у C/C++

У попередніх лабораторних роботах переважно використовувались базові типи даних (`char`, `int`, `long`, `float`, `double` тощо), які є стандартними для мов C/C++. Іноді також використовувались специфічні типи даних, визначені всередині стандартних бібліотек C/C++, наприклад, тип `size_t`. В найпростіших програмах на C/C++ базових типів даних та типів зі стандартних бібліотек може бути достатньо для виконання всіх потрібних дій. Проте в складних програмах на C/C++ цих типів даних, як правило, недостатньо для максимально ефективної обробки даних. Дуже часто дані мають зберігатись у вигляді сукупностей величин різного типу – тільки тоді алгоритм їх обробки буде максимально ефективним. Враховуючи це, мова C/C++ дозволяє програмісту створювати або визначати нові – *свої власні* – типи даних. Ці типи даних називаються ***структурованими***, оскільки мають певну внутрішню структуру у вигляді окремих елементів.

У мові C/C++ існує декілька варіантів структурованих типів – *переліки*, *структури*, *об'єднання* та *бітові поля*, які розглядаються в цій лабораторній роботі. Додатково у мові C++ існують такі структуровані типи як *класи*, які будуть розглядатись пізніше.

Структуровані типи не є базовими, вбудованими у C/C++[1]. Кожен з таких типів є по-своєму унікальним, незвичайним, тому їх можна вважати ***нестандартними*** типами даних.

Окрім структурованих типів даних, які є нестандартними «з самого початку», в мові C/C++ є можливість вводити «нестандартні» типи даних, які,

[1] Слід пам'ятати, що багато таких типів стають доступними лише при підключенні файлів заголовків стандартних бібліотек C/C++. Також, хоча самі структуровані типи не є базовими для мови C/C++, але власне механізм їх створення є невід'ємним елементом мови C/C++.

насправді, є аналогами вже визначених типів (як вбудованих, так і структурованих). Це відбувається за допомогою використання ключового слова `typedef` за наступним правилом:

```
typedef Тип Новий_тип;
```

В цьому рядку `Тип` – це ім'я вже існуючого (відомого) типу, а `Новий_тип` – це нове ім'я типу, яке може використовуватись як псевдонім (аліас) `Типу`.

Введення нового типу за допомогою `typedef` не змінює суті типу, його властивостей, проте дозволяє записувати код програми у більш зрозумілому вигляді. Такий підхід широко використовується в стандартних бібліотеках C/C++, при створенні програм для різних операційних систем тощо. Наприклад, наступний фрагмент коду вводить нові типи `UCHAR`, `UINT`, `ULONG`, які є псевдонімами відомих типів `unsigned char`, `unsigned int`, `unsigned long`:

```
typedef unsigned char UCHAR;
typedef unsigned int  UINT;
typedef unsigned long ULONG;
```

Враховуючи це, в програмі можна замість `unsigned char` використовувати тип `UCHAR`, замість `unsigned int` – тип `UINT` тощо.

Переліки або зчислені типи (enum)

Мова C/C++ дозволяє програмістам створювати свої власні типи даних. Одним із таких типів даних є *перелік* (enumerated type) або *тип даних, що перелічуються, зчислений тип даних*.

Перелік – це набір іменованих цілих чисел, визначений за допомогою ключового слова `enum`. Значення змінних цього типу є символьними константами, пов'язаними з певними цілими числами. Ці цілі числа за замовчуванням утворюють зростаючу послідовність починаючи з нуля з кроком 1. В момент оголошення переліку програміст може задавати/змінювати цілі числа, асоційовані з різними символьними константами. Оголошення переліку має наступний синтаксис:

```
enum Ім'я_переліку
{
   Ім'я_елементу_1 = Значення_1,
   Ім'я_елементу_2 = Значення_2,
```

```
    //...
    Ім'я_елементу_N = Значення_N
};
```

`Ім'я_переліку` є унікальним ідентифікатором, який визначає *конкретне ім'я типу* переліку. В програмі можна мати необмежену кількість переліків, які мають між собою відрізнятися ім'ям свого типу – `Ім'ям_переліку`. З точки зору компілятора `Ім'я_переліку` буде сприйматись як новий тип даних, який нічим не кращий і не гірший за вбудовані типи даних, такі як `int`, `char` тощо.

`Ім'я_елементу_1`, `Ім'я_елементу_2`, … `Ім'я_елементу_N` – це елементи – символьні константи, визначені всередині переліку. Їх кількість може бути довільною, зокрема, можна навіть оголосити перелік без будь яких елементів, наприклад, як `enum EmptyEnum{};` і, тим не менш, компілятор буде сприймати `EmptyEnum` як окремий новий тип. Імена елементів `Ім'я_елементу_1`, `Ім'я_елементу_2`, … `Ім'я_елементу_N` мають бути дозволеними ідентифікаторами, унікальними в межах всієї програми[1]. З кожним з цих елементів пов'язується деяке ціле число. Це число можна вказувати явним чином як `Значення_1`, `Значення_2`, … `Значення_N`, за допомогою оператора присвоювання `=`. Якщо ж відповідні значення не вказані, компілятор сам автоматично призначає кожному елементу переліку певне числове значення. Ці значення розраховуються за простим правилом: це значення попереднього елементу + 1, причому самий перший елемент в переліку за замовчуванням пов'язується із значенням 0.

Розглянемо декілька прикладів оголошення переліків:

```
enum Colors
{
    RED,
    BROWN,
    GRAY,
    WHITE,
    ORANGE
};
```

В цьому прикладі вводиться новий тип – перелік `Colors`. Після назви типу `Colors`, у фігурних дужках `{}`, перерахуються *всі допустимі* значення для змінної такого типу – символьні константи[2], які відокремлюються комами.

---

[1] Елементи переліків знаходяться в єдиному просторі імен. Отже, імена елементів в переліках не можуть повторюватися, навіть якщо відповідні елементи належать різним перелікам.

[2] Оскільки символьні константи асоціюються з цілими числами, насправді, в переліку зберігаються саме цілі числа, доступ до яких отримується через звернення до відповідних символьних констант.

Після останнього елементу ORANGE, згідно синтаксису C/C++, кома не ставиться. А після завершення опису всього типу ставиться крапка з комою.

У наведеному переліку Colors символьні константи задаються, але явно не пов'язуються з цілими числами за допомогою конструкції = Значення. Цілі числа, асоційовані з елементами переліку, автоматично призначає компілятор. При цьому RED стає асоційованим зі значенням 0, BROWN – 1, GRAY – 2, WHITE – 3 і ORANGE – 4. За необхідності можна змінити цілі числа, пов'язані з символьними константами, наприклад, таким чином:

```
enum Colors
{
     RED       = 10,
     BROWN     = 20,
     GRAY,
     WHITE     = 50,
     ORANGE
};
```

У цьому прикладі величині Colors::GRAY буде відповідати значення 21, а величині Colors::ORANGE – значення 51 (тобто значення, на одиницю більші за значення попередніх елементів).

Для звернення до певного елементу переліку слід вказати ім'я (тип) переліку, використати ***оператор дозволу видимості*** :: та вказати потрібне ім'я елементу переліку. Наприклад, Colors::ORANGE означає елемент ORANGE всередині переліку Colors.

Будь-які операції над елементами переліку виконуються як операції над *сталими цілими числами*. Наприклад, при спробі вивести на екран значення елементу переліку або значення змінної, пов'язаної з переліком, буде відображатися ціле число, а не ідентифікатор елемента (одна з символьних констант).

Із переліком можна працювати як з набором цілих чисел. Зокрема, символьні константи з переліку (у внутрішньому представленні – цілі числа) зручно використовувати для перевірки умов за допомогою операторів if, if-else або в операторі вибору switch. В останньому випадку у списку констант для case якраз і вказуються символьні константи, наведені в переліку. Наприклад для деякої змінної color типу int можна записати наступний фрагмент коду:

```
switch( color )
{
     case Colors::RED:
     {
          // Деякій оператор 1.
          break;
```

```
    }
    case Colors::WHITE:
    {
        // Деякій оператор 2.
        break;
    }
    default: break;
}
```

Тут значення змінної `color` порівнюється з константами `Colors::RED`, `Colors::WHITE` тощо. При співпадінні значення `color` з однією із цих констант виконуються деякі оператори 1 або 2. Якщо співпадінь знайдено не було оператор `switch` просто завершується.

Як видно з наведеного прикладу, переліки зручно використовувати як набір логічно зв'язаних між собою числових значень.

СТРУКТУРИ (STRUCT)

Структура – це агрегатний (складний, комплексний) тип даних, який складається з визначеної скінченної кількості елементів довільного типу. Відповідні елементи називаються ***членами*** або ***полями*** структури.

У мові C/C++ структури, в певному сенсі, є розвитком поняття масиву (див. лабораторну роботу № 4). Як і масиви структури можуть містити в собі довільну кількість елементів. Проте, на відміну від масивів, які є сукупністю змінних (даних) *одного* типу, структури дозволяють об'єднувати в єдине ціле – в єдиний новий тип – змінні (поля) *різних* типів.

Оголошення та визначення структур

Синтаксис оголошення структури має вигляд:

```
struct Ім'я_Структури
{
    Тип1 Ім'я1;
    Тип2 Ім'я2;
    //...
    ТипN Ім'яN;
};
```

`Ім'я_Структури` є унікальним ідентифікатором, який є ім'ям нового

типу, пов'язаного із структурою. Елементи `Тип1 Ім'я1`, `Тип2 Ім'я2`, … `ТипN Ім'яN` визначають поля структури. Кожне поле задається своїм типом (`Тип1`, `Тип2`, …) та ім'ям (`Ім'я1`, `Ім'я2`, …).

Навіть, якщо структура не містить жодного поля, вона все одно буде вважатись деяким новим унікальним типом даних з ім'ям `Ім'я_Структури`.

Типи, які використовуються для полів структури, мають бути визначені (відомі) на момент оголошення структури. Це можуть бути не тільки вбудовані типи, але й вказівники, масиви, переліки, інші структури, тощо. Однак, в мові C/C++ заборонено оголошувати і створювати структури, елементами яких є самі ці структури. Тобто створення структур виду `struct X{ X x; /* ... */ };`, де `X` – деякий тип, а `X x` – поле типу `X` з іменем `x`, не дозволяється! Проте створення структур, в яких присутні вказівники на самі ці структури, є дозволеним! Наприклад, наступне оголошення структури є коректним: `struct X{ X* px; /* ... */ X* px2; };`

Імена полів мають бути дозволеними ідентифікаторами, унікальними *лише в межах даної структури*. На відміну від переліків, де всі елементи повинні мати унікальні імена в межах всієї програми, для структур унікальність імен вимагається лише в межах самої структури. Отже, різні структури (різного типу) можуть мати поля з однаковими іменами та/або типами.

Розглянемо приклади оголошення структур. Почнемо з структури `Student`, яка може зберігати інформацію про студента університету:

```
struct Student
{
    // Прізвище Ім'я По батькові.
    char   Name[64];
    // Дата народження: Д-М-РР.
    UINT   DoB;
    // Курс.
    int    YearOfStudy;
    // Група.
    char   Group[16];
    // Оцінка = середній бал за сесію.
    double Grade;
};
```

Полями структури є символьний масив `Name`, який зберігає інформацію про прізвище, ім'я та по батькові студента; змінна цілого типу знаку (`typedef unsigned int UINT;`) `DoB`, в якій зашифрована дата народження студента (день зберігається в самому старшому байті, далі – місяць, молодші 2 байти – рік). Курс, на якому навчається студент, зберігається в полі `YearOfStudy`; група студента задається символьним рядком, що зберігається в полі `Group`. Нарешті, середній бал студента зберігається в останньому полі

`Grade` типу `double`.

За допомогою структури можна реалізувати збереження даних про комплексне число:

```
struct Complex
{
    double re, im;
};
```

Поля `re` та `im` цієї структури зберігають інформацію про дійсну та уявну частини комплексного числа. Зазначимо, що при оголошенні структур однотипні поля можна записувати в рядок, через кому, подібно тому як це робиться для однотипних змінних.

Для компілятора оголошення структури означає введення нового типу даних. Наведені вище приклади, де оголошуються структури `Student` та `Complex`, *лише інформують* компілятор про наявність двох нових типів даних – `Student` та `Complex`. Пам'ять під змінні відповідного типу поки що не виділяється! Щоб почати використовувати ці типи, необхідно спочатку оголосити змінні цих типів. Це здійснюється за загальними правилами, наприклад:

```
Student stud1, stud2;
Complex c1, c2;
```

Тут оголошуються змінні або, іноді говорять, ***екземпляри*** `stud1` та `stud2` типу `Student`, а також змінні (екземпляри структур) `c1` та `c2` типу `Complex`. Дане оголошення записане в *стилі C++*, коли перед іменем нововведених типів не потрібно вказувати ключове слово `struct`. Однак, якщо компілятор перебуває в режимі компіляції коду C, при записі імені типу структури може знадобитись вказати додаткове ключове слово `struct`. У цьому випадку оголошення змінних має відбуватись так: `struct Student stud1, stud2;`, `struct Complex c1, c2;`. З огляду на цю особливість синтаксису мови C, використання режиму компіляції коду як коду C++ дозволяє спростити написання програм. Саме тому переважна більшість програм, навіть написаних виключно мовою C, зазвичай компілюються в режимі C++.

Подібно змінним стандартних типів і масивам, змінні, пов'язані зі структурами, можна ***визначати***. При цьому змінна не тільки оголошується, але ще й одночасно ініціалізується певним «значенням». Оскільки поля структури можуть мати різні типи, відповідне «значення» також повинне містити в собі різнорідні величини (різних типів).

Визначення змінних структури трохи нагадує визначення елементів масиву. Воно має наступний синтаксис:

```
Тип Змінна = {Зн_поля_1, Зн_поля_2, /*...*/ Зн_поля_N};
```

Так само, як і під час визначення звичайної змінної, при визначенні структури деякого `Типу` слід написати оператор присвоювання `=` і потім у фігурних дужках `{}` задати конкретні значення для полів структури. Порядок запису цих значень відповідає порядку запису полів у структурі. Тобто значення `Зн_поля_1` буде використане для ініціалізації першого поля структури, `Зн_поля_2` – другого, `Зн_поля_N` – останнього. Якщо значення ініціалізації для деяких полів не задані, відповідні поля ініціалізуються значеннями за замовчуванням (як правило, це 0). Нижче наведено приклад ініціалізації змінної `stud1` типу `Student`, який був введений раніше:

```
Student stud1 =
{
    "Мельник Оксана Юріївна",
    UINT(15)<<24 | UINT(2)<<16 | UINT(2002),
    4,
    "Група 1",
    96.345
};
```

Текстовий рядок `"Мельник Оксана Юріївна"` записується в поле `Name`, яке є символьним масивом. У поле `DoB` записується дата народження – 15 лютого 2002 року. Студентка навчається на 4 курсі, в першій групі, відповідні значення `4` та `"Група 1"` записуються в поля `YearOfStudy` та `Group`. Середній бал студентки задається останнім значенням `96.345`. Якщо, наприклад, це останнє значення та кому після `"Група 1"` прибрати, поле `Grade` буде ініціалізоване значенням за замовчуванням і, зазвичай, буде містити `0.0`.

Доступ до полів структур. Зміна значень полів

Значення змінних, пов'язаних із структурами, можна зчитувати і змінювати в процесі роботи програми. Це відбувається шляхом зчитування або зміни вмісту полів цих структур. Ці операції по суті ідентичні операціям зі звичайними змінними стандартних типів, проте мають трохи складніший синтаксис.

Для безпосереднього доступу до поля *екземпляра* (змінної типу) структури використовується спеціальна операція доступу `.`, яку іноді називають «операція крапка». Якщо є екземпляр `x` деякої структури, в якій є поле з іменем `y`, фрагмент коду `x.y` означає «поле `y` всередині екземпляру структури `x`». Наприклад, для визначеного вище екземпляру `stud1` типу `Student`, `stud1.Name` означає звернення до поля `Name`, а `stud1.Grade` – звернення

до поля `Grade`. За умови належного звернення, значення, що зберігаються всередині полів, можуть зчитуватись і перезаписуватись за загальними правилами, так само як для звичайних змінних. Наприклад, фрагмент коду

```
printf(stud1.Name);
```

дозволяє вивести на консоль вміст поля `Name`, що знаходиться всередині змінної-екземпляра структури `stud1`. А фрагмент коду

```
stud1.Grade = 100.0;
```

змінить вміст поля `Grade` для цієї змінної-екземпляра на 100.0 (тобто студентка Мельник Оксана Юріївна стане «круглою відмінницею»).

Іншим способом доступу до полів певного екземпляру структури є доступ через вказівник `p` на цей екземпляр (змінну). Для цього використовується операція доступу `->`, яку іноді називають «операцією стрілка». Синтаксис використання цієї операції наступний:

```
p->Поле
```

Наведений фрагмент інтерпретується як звернення до `Поля` всередині екземпляру структури, на який вказує вказівник `p`. Записаний код є більш простим аналогом коду `(*p).Поле`. У формі запису `(*p).Поле` дужки навколо `*p` є обов'язковими, оскільки «операція крапка» має більший пріоритет, аніж операція розіменування вказівника. При використанні операції `->` фрагмент коду записується без дужок, операцій `*` та `.` і тому виглядає дещо простіше.

За аналогією з раніше наведеними прикладами, код, що використовує операцію `->` для доступу до полів `stud1` та зміни їх значень, може мати вигляд:

```
Student* p = &stud1; // Ініціалізувати вказівник.
printf(p->Name);     // Вивести значення Name.
p->Grade = 100.0;    // Змінити значення Grade.
```

Операції зі структурами

Операції над екземплярами структур мають виконуватись як над звичайними змінними. Проте структури мають складну будову (можуть мати різнотипні елементи-поля), тому багато операцій (`+`, `-`, `*` тощо), дозволених для

змінних вбудованих типів, для змінних-структур є невизначеними[1]. За замовчуванням компілятор, як правило, реалізує лише операцію присвоювання = і операцію визначення адреси &.

Операція присвоювання = за замовчуванням реалізується як операція побайтового копіювання вмісту одного екземпляру структури в інший екземпляр тієї ж структури. Наприклад, фрагмент коду `stud2 = stud1;` приведе до копіювання даних з екземпляра структури `stud1` типу `Student` в екземпляр `stud2` того ж типу `Student`. Зазначимо, що в багатьох випадках такий код виявляється досить ефективним і надійним. Однак, якщо всередині структур є поле або поля, пов'язані з вказівниками, таке просте копіювання даних іноді може призводити до виникнення збоїв, нестабільної роботи програми. Щоб уникнути цього, можна самостійно реалізувати перевантажений оператор присвоювання для екземплярів тих структур, в яких активно використовуються вказівники (це буде продемонстровано в наступних лабораторних роботах).

Операція визначення адреси & реалізується за тими же правилами, що й операція визначення адреси для змінних стандартних типів. Ця операція ефективно працює для будь-яких структур і немає жодних «побічних» ефектів, чи застережень. При її використанні виявляється цікава, хоча й очевидна, особливість: адреса екземпляра структури співпадає з адресою першого поля структури. Так, для раніше введеної структури типу `Student` адреса екземпляру `stud1` цієї структури виявляється такою ж самою як і адреса поля `Name` цього екземпляру. А враховуючи те, що поле `Name` є символьним масивом, виявляється, що всі три адреси `&stud1`, `&stud1.Name` та `&stud1.Name[0]` є однаковими – вони вказують на одну і ту ж саму комірку пам'яті.

Масиви структур

У мові C/C++ є можливість створювати масиви структур. Це відбувається за стандартними правилами – тип даних масиву вказується як тип даних структури. Наприклад, наступні рядки коду оголошують масив з 30 комплексних чисел типу `Complex` і масив з 1000 записів типу `Student` – тих типів структур, які були введені раніше:

```
Complex ca[30];
Student students[1000];
```

Для доступу до полів структур-елементів цих масивів використовуються ті ж самі правила, як і при роботі з масивами стандартних типів. Наприклад, щоб змінити дійсну та уявну частини комплексного числа, яке зберігається в

[1] Деякі з таких операцій можна реалізувати, використовуючи технологію перевантаження операторів, яка буде розглядатись пізніше.

елементі масиву з індексом 20, використовується фрагмент коду:

```
ca[20].re = 1.0;
ca[20].im = -2.0;
```

Звернення `ca[20]` сприймається компілятором як елемент масиву з індексом 20 – тобто екземпляр структури типу `Complex`. Далі до цього екземпляру застосовується операція `.`, яка дозволяє отримати доступ до полів `re` та `im` цього екземпляру структури і змінити їх, присвоївши їм нові значення `1.0` та `-2.0`.

Фрагмент коду, що виконує аналогічну роботу, але використовує замість `.` «операцію стрілка» `->`, виглядатиме так:

```
(ca+20)->re = 1.0;
(ca+20)->im = -2.0;
```

До цього часу розглядались лише одновимірні масиви структур. Проте мова C/C++ дозволяє програмісту створювати масиви структур будь-якої скінченної розмірності. Правила роботи з такими масивами нічим не відрізняються від правил роботи з багатовимірними масивами даних стандартних типів (див. лабораторну роботу № 4).

Структури як аргументи і результат роботи функцій

Функції в програмах мовою C/C++ можуть мати аргументи у вигляді як самих структур, так і вказівників та посилань на структури, масивів структур. Функції також можуть повертати значення типу структури, вказівника та посилання на структуру, масиву структур.

З точки зору процедури передачі параметрів у функцію, повернення функцією результату своєї роботи, робота зі структурами принципово нічим не відрізняється від роботи з даними стандартних типів. Слід, однак, пам’ятати, що структури часто займають достатньо великий об’єм пам’яті, тому передавати їх безпосередньо у функцію може бути нераціонально (це потребуватиме великого об’єму стеку та/або значної кількості операцій з ним). Так само нераціональним може бути повернення з функції результату у вигляді структури[1]. Щоб обійти це «вузьке місце», функції, які працюють зі структурами, часто приймають і повертають не самі структури, а *вказівники* або *посилання на ці структури*. Останній спосіб є найбільш поширеним, оскільки дозволяє не використовувати операцію розіменування вказівника `*`, що трохи спрощує

[1] Безпосередньо приймати і повертати значення типу структури може бути прийнятним для структур досить малого розміру, як правило, не більше 8/16 байт для 32/64-бітних програм.

вигляд тексту програми.

При використанні посилань на структури вживаються дві форми запису посилань: *звичайні* (`Тип_структури&`) і *сталі* (`const Тип_структури&`) посилання. Звичайні посилання дозволяють як зчитувати, так і змінювати вміст структури, пов'язаної з посиланням. Сталі посилання, задані за допомогою ключового слова `const`, інформують компілятор про те, що об'єкт, на який вказує посилання, є сталим і не підлягає модифікації (його вміст не може змінюватись). Якщо компілятор зустрічає фрагмент коду, в якому відбувається зміна об'єкта через стале посилання, він генерує помилку компіляції. Таким чином, сталі посилання можуть використовуватись лише для зчитування даних деякої структури, а звичайні посилання (без ключового слова `const`) можна використовувати як для читання, так і для зміни даних структури. Лістинг 7.1 демонструє приклад декількох функцій для роботи зі структурами типу `Complex`, які описують комплексні числа.

Лістинг 7.1:

```
 1: #include <stdio.h>
 2: #include <math.h>
 3:
 4: struct Complex
 5: {
 6:     double re, im;
 7: };
 8:
 9: Complex cAdd( const Complex& a, const Complex& b )
10: {
11:     Complex c;
12:     c.re = a.re + b.re;
13:     c.im = a.im + b.im;
14:     return c;
15: }
16:
17: double cArg( const Complex& a )
18: {
19:     return atan2(a.im, a.re);
20: }
21:
22: Complex& cIncBy( Complex& X, const Complex& Y )
23: {
24:     X.re += Y.re;
25:     X.im += Y.im;
26:     return X;
27: }
28:
```

```
29: double cModule( const Complex& a )
30: {
31:     return sqrt(a.re*a.re + a.im*a.im);
32: }
33:
34: void cPrint( const Complex& a )
35: {
36:     printf("Complex( %f , %f )", a.re, a.im);
37: }
38:
39: int main( void )
40: {
41:     Complex a = { -1.0, 1.0};
42:     Complex b = { 3.0, 7.0 };
43:
44:     printf("a == ");
45:     cPrint(a);
46:     printf("\tb == ");
47:     cPrint(b);
48:     printf("\n\n");
49:
50:     Complex c = cAdd(a, b);
51:     printf("c == a + b == ");
52:     cPrint(c);
53:     printf("\n\n");
54:
55:     printf("c == c + a == ");
56:     cIncBy(c, a);
57:     cPrint(c);
58:     printf("\n\n");
59:
60:     printf("Module of c is %f , argument is
61: %f\n\n\n", cModule(c), cArg(c));
62:
63:     return 0;
64: }
```

У лістингу визначаються функції `cAdd()`, `cArg()`, `cIncBy()`, `cModule()` та `cPrint()`, які вміють працювати зі структурами типу `Complex`.

Функція `cAdd()` реалізує додавання двох комплексних чисел (рядки 9-15). Вона приймає два аргументи типу `const Complex&`, дані яких доступні тільки для читання. Всередині тіла функції створюється проміжна локальна змінна `Complex c`, у яку записується результат операції додавання

комплексних чисел (рядки 12, 13), переданих через сталі посилання `a` та `b`.

Доступ до дійсної та уявної частини комплексних чисел, пов'язаних з полями `re` та `im` у структурі `Complex`, здійснюється за допомогою операції доступу `.`. Тобто `a.re` інтерпретується як дійсна частина комплексного числа, на яке вказує стале посилання `a`, `b.im` – уявна частина комплексного числа, на яке вказує стале посилання `b` тощо.

При завершенні функції `cAdd()` результат її роботи повертається безпосередньо у вигляді екземпляру структури `c`. Зазначимо, що дана функція не може просто повернути результат у вигляді посилання `Complex&` на `c`, оскільки `c` – локальна змінна, область дії і час життя якої обмежені тілом функції `cAdd()`. Коли ця функція завершиться, екземпляр структури `c` стане недійсним, а отже недійсним буде і посилання на `c`.

Функції `cArg()` та `cModule()` розраховують аргумент та модуль комплексного числа, переданого їм через стале посилання. Функція `cPrint()` виводить на консоль вміст комплексного числа за допомогою `printf()`.

Функція `cIncBy()` реалізує приріст першого переданого їй комплексного числа на величину другого комплексного числа, заданих через звичайне посилання `X` та стале посилання `Y`. Тобто фактично виконує операцію `X += Y;`. Тіло цієї функції включає в себе приріст дійсної (`X.re += Y.re;`) та уявної (`X.im += Y.im;`) частини комплексного числа, заданого звичайним(!) посиланням `X`. Результат роботи цієї функції повертається у вигляді посилання на `X`, яке було визначене ще до початку роботи функції, а значить залишиться коректним і після завершення її роботи.

Всі перелічені вище функції викликаються в тілі функції `main()`, а результат їх роботи виводиться на консоль (рис. 25).

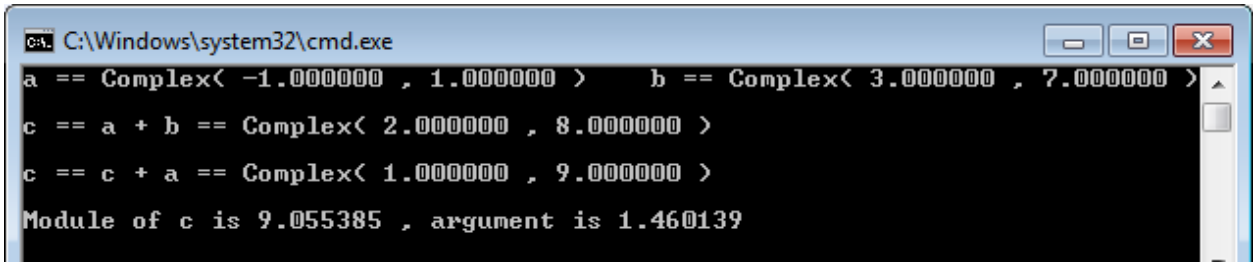


Рис. 25

Об'єднання (union)

Об'єднання – це структура, усі поля якої розділяють/займають одну і ту ж саму область пам'яті. Це означає, що всі поля об'єднання перекриваються між собою. Отже, інтерпретація вмісту об'єднання залежить від того, до якого його поля відбувається звернення. Розмір об'єднання дорівнює розміру його

найбільшого поля.

Оголошення об'єднання подібне до оголошення структур, але замість ключового слова `struct` використовується ключове слово `union`. Наприклад, об'єднання `int_uint_or_float` може містити або ціле число без знаку (поле `ui`), або ціле число зі знаком (поле `i`), або дійсне число типу `float` (поле `f`):

```
union int_uint_or_float
{
    int           i;
    unsigned int ui;
    float         f;
};
```

Всі три поля `int_uint_or_float` мають однаковий розмір (4 байти), тому розмір об'єднання також дорівнює 4 байтам. Якщо ж, наприклад, змінити тип `float` в оголошенні об'єднання на `double`, розмір об'єднання зміниться – він вже буде дорівнювати розміру найбільшого поля `double f`, тобто `sizeof(double)` або 8 байт.

Оскільки всі поля `i`, `ui` та `f` об'єднання `int_uint_or_float` розділяють одну і ту ж саму область пам'яті, запис даних в одне з цих полів автоматично змінює дані, що зберігались в інших полях. Ця властивість може бути використана для автоматичної конвертації даних з однієї форми в іншу. Наприклад, наведене вище об'єднання `int_uint_or_float` можна використати для перетворення величин з типу `float` в еквівалентні величини типу `int` або `unsigned int`. Такі перетворення по суті дають різні інтерпретації одного і того ж набору бінарних даних як величин різного типу. У лістингу 7.2 наведена програма, яка використовує цю властивість об'єднань.

Лістинг 7.2:

```
 1: #include <stdio.h>
 2:
 3: union int_uint_or_float
 4: {
 5:     int           i;
 6:     unsigned int ui;
 7:     float         f;
 8: };
 9:
10: int main( void )
11: {
12:     int_uint_or_float Union;
13:
```

```
14:      Union.f = 1.0f;
15:
16:      printf("float \t\t = %f\n", Union.f);
17:      printf("int \t\t = %d\n", Union.i);
18:      printf("unsigned int \t = %u = 0x%X\n\n",
19: Union.ui, Union.ui);
20:
21:      Union.f = -3.1415926535f;
22:
23:      printf("float \t\t = %f\n", Union.f);
24:      printf("int \t\t = %d\n", Union.i);
25:      printf("unsigned int \t = %u = 0x%X\n\n\n",
26: Union.ui, Union.ui);
27:
28:      return 0;
29: }
```

У програмі використовується раніше введене об'єднання `int_uint_or_float` (рядки 3-8). В тілі функції `main()` визначається екземпляр (змінна) `Union` відповідного типу (рядок 12).

Спочатку в поле `f` об'єднання `Union` записується число `1.0f` (рядок 14). Далі, за допомогою `printf()`, виводиться вміст усіх трьох полів (рядки 16-19).

Після цього в поле `Union.f` записується наближене значення числа –π і результат знову виводиться в різних формах за допомогою `printf()` (рядки 21, 23-26). Результат роботи програми зображено на рис. 26.

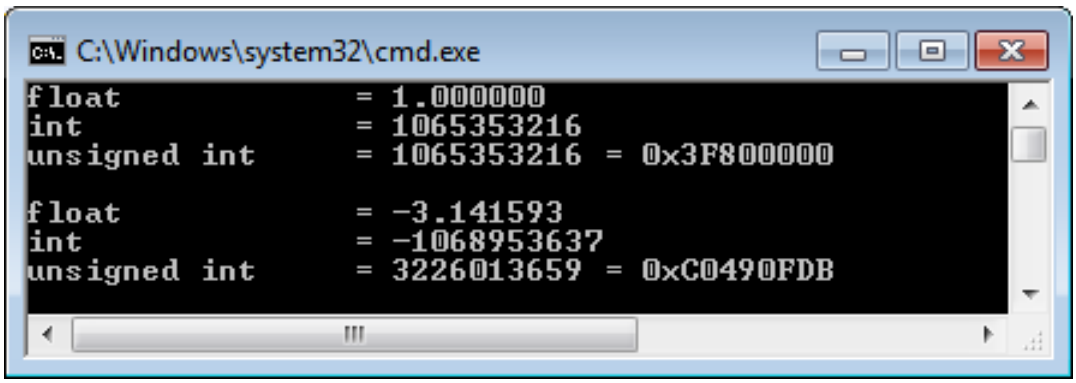


Рис. 26

Як видно з рис. 26 один і той же самий набір з 32 бітів може інтерпретуватись зовсім по-різному, залежно від того з яким типом даних планується працювати.

Об'єднання також можна використовувати для збереження даних в деяких еквівалентних формах. Наприклад, температуру можна вимірювати в градусах Цельсія, градусах Фаренгейта або в кельвінах; масу у кілограмах або

фунтах; енергію в джоулях або електронвольтах тощо. Часто також використовується запис координат в абсолютних одиницях або у відносних (заданих відносно деякої точки), що легко реалізувати наступним чином:

```
union Coord
{
     // Відносні координати можуть бути <> 0.
     int x;
     // Абсолютні координати завжди >= 0.
     unsigned int X;
};
```

Оскільки об'єднання – це специфічний вид структури, до об'єднань можна застосовувати ті ж самі операції що і для структур. При цьому, аналогічно до структур, операції «крапка» `.` та «стрілка» `->` використовуються для доступу до полів об'єднання. Використання операції `.` демонструється в лістингу 7.2 для звернення до полів `f`, `i`, `ui` об'єднання `Union`: `Union.f`, `Union.i`, `Union.ui`.

Об'єднання не мають стандартної конструкції для визначення їх «поточного стану», тобто визначення того їх елементу, який явно задавався в програмі. Це означає, що хоча в програмі з лістингу 7.2 задавалось значення елементу `Union.f`, визначити, яке саме поле задавалось в `Union`, лише за вмістом цього об'єднання *неможливо*! Отже, перевірити, яке саме поле об'єднання в довільний момент часу є «активним» також не можливо. Контроль за цим покладається на самого програміста[1].

Бітові поля

Бітове поле (англ. bit field) – це структура, яка містить поля довжиною в один або кілька (до 32/64) бітів. Поля бітового поля повинні мати тип `int`, `short`, `signed short/int`, `unsigned short/int` (останній варіант, який записують як `unsigned`, зустрічається найбільш часто). Синтаксис оголошення бітового поля наступний:

```
struct Ім'я_Бітового_поля
{
     Тип Ім'я1 : Розмір1;
```

[1] Для реалізації такого контролю, зазвичай, вводять додаткову змінну, в якій зберігається інформація про «активне» поле об'єднання. Ця змінна і об'єднання можуть бути елементами однієї структури. Тоді за значенням змінної можна визначити до якого поля об'єднання потрібно звернутись.

```
    Тип Ім'я2 : Розмір2;
    //...
    Тип Ім'яN : РозмірN;
};
```

`Ім'я_Бітового_поля` є унікальним іменем типу бітового поля.

`Тип` є базовим типом, який використовується для побудови бітового поля. Це є один з дозволених цілочислених типів, згаданих вище (`signed`/`unsigned`, `short`/`int`). Деякі компілятори C/C++ дозволяють створювати бітові поля і на основі інших типів, наприклад, `char`.

`Ім'я1`, `Ім'я2`, … `Ім'яN`, є іменами полів, що мають довжини в бітах `Розмір1`, `Розмір2`, …, `РозмірN`. Розмір поля є обов'язковим параметром. Цей розмір є натуральним числом, що змінюється в межах 1 – 32/64 біти для 32/64-бітних систем. На відміну від структур, в бітових полях імена всіх полів `Ім'я1`, `Ім'я2`, … `Ім'яN` є необов'язковими. Якщо ім'я якогось поля відсутнє, таке безіменне поле буде фізично існувати всередині бітового поля, проте звернутися до цього безіменного поля буде неможливо. Безіменні поля часто використовують для позначення тих груп бітів, які є несуттєвими, не використовуються. Іноді безіменні поля використовують як певний елемент-розділювач для різних груп бітів.

Адресація пам'яті в сучасних комп'ютерах реалізується шляхом адресації окремих байтів. При цьому визначення адреси блоку даних, який має розмір, менший за 1 байт, не дозволяється. Тому при роботі з бітовими полями визначення адреси полів у бітовому полі за допомогою оператора `&` заборонено.

На різних комп'ютерах з різними типами процесорів та ОС, фізичне розташування байтів всередині машинного слова (2 байти), слів всередині подвійних слів (4 байти) тощо, може бути неоднаковим. Відповідно, на різних комп'ютерах одне і те ж саме бітове поле може відображатись в пам'ять порізному. Це означає, що використання бітових полів є системно-залежним, а значить код, де використовуються бітові поля, може бути неуніверсальним. Тип не менш, робота з бітовими полями іноді є більш простою, ніж використання побітових операцій, які розглядались в лабораторній роботі № 6. Приклад, роботи з бітовим полем розглянуто в лістингу 7.3:

Лістинг 7.3:

```
 1: #include <stdio.h>
 2:
 3: typedef unsigned short USHORT;
 4:
 5: struct BitTime   // total size is 16 bit only
 6: {
 7:     USHORT Hour : 5;// 0-23
```

```
 8:      USHORT Mins : 6;// 0-59
 9:      USHORT IsAM : 1;// AM / PM
10:      USHORT      : 4;// unused
11: };
12:
13: int main( void )
14: {
15:      BitTime bt;
16:      bt.Hour = 8; // 8:50 ранку.
17:      bt.Mins = 50;
18:      bt.IsAM = 1;
19:
20:      printf(
21:           "Time is %2u:%2u\n\n",
22:           bt.IsAM ? bt.Hour : bt.Hour + 12,
23:           bt.Mins
24:      );
25:
26:      bt.IsAM = 0; // Після опівдня.
27:      printf(
28:           "Time is %2u:%2u\n\n",
29:           bt.IsAM ? bt.Hour : bt.Hour + 12,
30:           bt.Mins
31:      );
32:
33:      return 0;
34: }
```

У рядку 3 за допомогою `typedef unsigned short USHORT;` визначається тип `USHORT`, який є аналогом типу `unsigned short`. Використовуючи цей тип, в рядках 5-11 оголошується бітове поле `BitTime`, здатне зберігати дані про час (годину та хвилини) в 12-годинному форматі. Поле `IsAM`, довжиною 1 біт, визначає чи час заданий відносно 0:00 (біт встановлено), чи заданий він відносно 12:00 (біт скинуто). Для збереження годин використовується поле `Hour`, довжиною 5 біт, а для хвилин – поле `Mins`, довжиною 6 біт. Останнє поле є безіменним і має довжину 4 біти, в результаті чого все бітове поле має сумарну довжину 5 + 6 + 1 + 4 = 16 біт.

У рядках 15-18 до бітового поля `BitTime bt` записується час 8:50 ранку за 24-годинним форматом. Інформація про цей час виводиться за допомогою `printf()` на консоль (рядки 20-24). Далі біт `IsAM` скидається і інформація про час виводиться на консоль знову (рядки 27-31). Щоб коректно виконати перетворення годин використовується тернарний умовний оператор `bt.IsAM ? bt.Hour : bt.Hour + 12` (рядок 29). Якщо «прапорець» `bt.IsAM` встановлений (дорівнює 1), виводяться години, збережені в полі

`bt.Hour`. Якщо ж «прапорець» `bt.IsAM` скинутий, години в 12-годинному форматі конвертуються в 24-годинний формат як `bt.Hour + 12`, що вже дає час 20:50.

Як видно з наведеного прикладу робота з елементами-полями всередині бітового поля є простішою, аніж реалізація різних побітових операцій, за допомогою яких можна змінювати/визначати вміст різних груп бітів всередині цілих чисел. При використанні бітових полів всю рутинну технічну роботу по виконанню таких дій компілятор C/C++ бере на себе.

Завдання

Варіант 1

1) Визначити перелік, в якому зберігаються різні типи компонент комп'ютера. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного компонента, інакше – виводити повідомлення, що компонент невідомий.
2) Визначити структурований тип даних, який описує комплексне число в алгебраїчній формі (дійсна та уявна частина). Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, аргументом якої є згаданий тип даних, і яка обчислює модуль та аргумент комплексного числа.
4) Створити функції які обчислюють суму, різницю, добуток та частку комплексних чисел.
5) Створити функції яка виводить на екран комплексне число в алгебраїчному форматі (наприклад, 6 + 8i).
6) Створити два довільних комплексних числа, виконати з ними операції, вказані у п. 3-5 та вивести результати на екран з відповідними коментарями.
7) Впорядкувати отримані в попередніх пунктах 6 комплексних чисел за зростанням модуля комплексного числа та вивести результат на екран.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з комплексними числами, де залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або в алгебраїчному, або в експоненціальному форматах.

Варіант 2

1) Визначити перелік, в якому зберігаються різні марки автомобілів. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння

функція має виводити на консоль назву відповідної марки автомобіля, інакше – виводити повідомлення, що марка автомобіля невідома.

2) Визначити структурований тип даних, який описує раціональне число, задаючи його чисельник та знаменник. Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, аргументом якої є згаданий тип даних, і яка переводить це число у форму з плаваючою крапкою.
4) Створити функції які обчислюють суму, різницю, добуток та частку описаних вище раціональних чисел.
5) Створити функції яка виводить на екран раціональне число у вигляді відношення чисельника до знаменника (наприклад, 123/456).
6) Створити два довільних раціональних числа, виконати з ними операції, вказані у п. 3-5 та вивести результати на екран з відповідними коментарями.
7) Впорядкувати отримані в попередніх пунктах 6 раціональних чисел за зростанням їх чисельника та вивести результат на екран.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з раціональними числами і числами з плаваючою крапкою, де залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або в раціональній формі, або в формі числа з плаваючою крапкою.

Варіант 3

1) Визначити перелік, в якому зберігаються різні частини тіла людини. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної частини тіла, інакше – виводити повідомлення про невідому частину тіла.
2) Визначити структурований тип даних, який описує точку у двовимірному просторі (через дві декартові координати). Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, яка отримавши аргументами дві довільні точки, обчислює відстань між ними та виводить результат на екран.
4) Створити функцію, яка, отримавши аргументами три довільні точки, визначає, чи лежить середня точка на одній прямій із двома крайніми, і виводить результат у вигляді відповідного коментаря.
5) Створити функцію, яка, отримавши аргументами три довільні точки, обчислює значення кута, вершиною якого є середня точка, а променями – відрізки, що з'єднують середню точку з двома крайніми, і виводить результат на екран.
6) Створити функцію, яка отримавши аргументами три довільні точки, обчислює площу трикутника з вершинами у вказаних точках, і виводить результат на екран.
7) Вибрати декілька довільних точок і продемонструвати для них результати

роботи функцій, вказаних у п. 3-6.

8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з точками на площині, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді декартових координат, або у вигляді полярних координат (радіус і кут).

Варіант 4

1) Визначити перелік, в якому зберігаються різні види квіткових рослин. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної рослини, інакше – виводити повідомлення, що рослина невідома.
2) Визначити структурований тип даних, який описує точку у тривимірному просторі (через три декартові координати). Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, яка, отримавши аргументами дві довільні точки, обчислює відстань між ними та виводить результат на екран.
4) Створити функцію, яка, отримавши аргументами три довільні точки, визначає, чи лежить середня точка на одній прямій із двома крайніми, і виводить результат у вигляді відповідного коментаря.
5) Створити функцію, яка, отримавши аргументами три довільні точки, обчислює значення кута, вершиною якого є середня точка, а променями – відрізки, що з’єднують середню точку з двома крайніми, і виводить результат на екран.
6) Створити функцію, яка, отримавши аргументами три довільні точки, обчислює площу трикутника з вершинами у вказаних точках, і виводить результат на екран.
7) Вибрати декілька довільних точок і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з точками в просторі, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді декартових координат, або у вигляді сферичних координат (радіус і два кути).

Варіант 5

1) Визначити перелік, в якому зберігаються різні типи геометричних фігур. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної фігури, інакше – виводити повідомлення, що фігура невідома.
2) Визначити структурований тип даних, який описує плоску фігуру, а саме трикутник. Трикутник задається тривимірними декартовими

координатами своїх вершин. Перейменувати цей тип за допомогою `typedef`.

3) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, перевіряє, чи дійсно всі точки фігури лежать в одній площині та виводить результат на екран.
4) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює периметр відповідної фігури і виводить результат на екран.
5) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює площу відповідної фігури і виводить результат на екран.
6) Додатково визначте структурований тип даних, який описує коло (задається координатами центра та радіусом) та виконайте для нього завдання п. 4, 5.
7) Вибрати декілька значень координат і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з трикутниками та колами, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді даних для трикутника, або у вигляді даних для кола.

Варіант 6

1) Визначити перелік, в якому зберігаються різні типи шахових фігур. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної шахової фігури, інакше – виводити повідомлення, що фігура невідома.
2) Визначити структурований тип даних, який описує плоску фігуру, а саме прямокутник. Прямокутник задається тривимірними декартовими координатами своїх вершин. Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, перевіряє, чи дійсно всі точки фігури лежать в одній площині та виводить результат на екран.
4) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює периметр відповідної фігури і виводить результат на екран.
5) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює площу відповідної фігури і виводить результат на екран.
6) Додатково визначте структурований тип даних, який описує коло (задається координатами центра та радіусом) та виконайте для нього завдання п. 4, 5.
7) Вибрати декілька значень координат і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для

роботи з прямокутниками та колами, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді даних для прямокутника, або у вигляді даних для кола.

Варіант 7

1) Визначити перелік, в якому зберігаються різні книги. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної книги, інакше – виводити повідомлення, що книга невідома.
2) Визначити структурований тип даних, який описує плоску фігуру, а саме паралелограм. Паралелограм задається тривимірними декартовими координатами своїх вершин. Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, перевіряє, чи дійсно всі точки фігури лежать в одній площині та виводить результат на екран.
4) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює периметр відповідної фігури і виводить результат на екран.
5) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює площу відповідної фігури і виводить результат на екран.
6) Додатково визначте структурований тип даних, який описує коло (задається координатами центра та радіусом) та виконайте для нього завдання п. 4, 5.
7) Вибрати декілька значень координат і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з паралелограмами та колами, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді даних для паралелограма, або у вигляді даних для кола.

Варіант 8

1) Визначити перелік, в якому зберігаються номінали різних гральних карт без урахування масті. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль номінал відповідної карти, інакше – виводити повідомлення, що карта невідома.
2) Визначити структурований тип даних, який описує плоску фігуру, а саме ромб. Ромб задається тривимірними декартовими координатами своїх вершин. Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, перевіряє, чи дійсно всі точки фігури лежать в одній площині та

виводить результат на екран.

4) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює периметр відповідної фігури і виводить результат на екран.
5) Створити функцію, яка, отримавши аргументом змінну вказаного вище типу, обчислює площу відповідної фігури і виводить результат на екран.
6) Додатково визначте структурований тип даних, який описує коло (задається координатами центра та радіусом) та виконайте для нього завдання п. 4, 5.
7) Вибрати декілька значень координат і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з ромбами та колами, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді даних для ромба, або у вигляді даних для кола.

Варіант 9

1) Визначити перелік, в якому зберігаються різні напрями науки. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного напряму, інакше – виводити повідомлення, що напрям невідомий.
2) Визначити структурований тип даних, який описує комплексне число в експоненціальній формі (модуль та фаза). Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, аргументом якої є згаданий тип даних, і яка обчислює дійсну та уявну частину комплексного числа.
4) Створити функції, які обчислюють суму, різницю, добуток та частку комплексних чисел.
5) Створити функцію, яка виводить на екран комплексне число в експоненціальному форматі (наприклад, 10×exp(0.3i)).
6) Створити два довільних комплексних числа, виконати з ними операції, вказані у п. 3, 4 та вивести результати на екран з відповідними коментарями.
7) Впорядкувати отримані в попередніх пунктах 6 комплексних чисел за зростанням фази комплексного числа та вивести результат на екран.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з комплексними числами, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або в експоненціальному, або в алгебраїчному форматах.

Варіант 10

1) Визначити перелік, в якому зберігаються різні типи компонент літака. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного компонента, інакше – виводити повідомлення, що компонент невідомий.
2) Визначити структурований тип даних, який описує точку на площині в полярних координатах. Перейменувати цей тип за допомогою `typedef`.
3) Створити функцію, аргументом якої є згаданий тип даних, яка обчислює та виводить на екран декартові координати точки.
4) Створити функцію, яка, отримавши в якості аргументів декартові координати точки, повертає об’єкт згаданого типу даних, що містить полярні координати даної точки.
5) Створити функції, які, отримавши в якості аргументів дві точки, обчислюють відстань між ними та скалярний добуток векторів, проведених із початку координат у дві вказані точки.
6) Створити функцію, яка обчислює полярні координати точки, що лежить рівно посередині між двома заданими.
7) Вибрати декілька значень координат і продемонструвати для них результати роботи функцій, вказаних у п. 3-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з точками на площині, де, залежно від значення «прапорця», довжиною 1 біт, дані б зберігались або у вигляді полярних координат, або у вигляді декартових координат.

Варіант 11

1) Визначити перелік, в якому зберігаються різні навчальні дисципліни з освітньої програми. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної дисципліни, інакше – виводити повідомлення, що дисципліна невідома.
2) Визначити структурований тип даних, який описує абонента у телефонній книзі. Тип повинен містити: номер телефону, ПІБ абонента, місто, в якому він проживає. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим номером телефону виводить на екран всю інформацію про відповідного абонента з масиву даних.
5) Написати функцію, яка за заданим текстовим символом виводить на екран прізвища всіх абонентів, що починаються з цього символу.

6) Написати функцію, яка самостійно аналізує масив даних і виводить на екран абонентів, які проживають в одному місті (якщо вони є). Якщо таких груп абонентів декілька, мають бути виведені дані всіх груп.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та `unsigned int`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 12

1) Визначити перелік, в якому зберігаються різні породи собак. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної породи собак, інакше – виводити повідомлення, що порода невідома.
2) Визначити структурований тип даних, який описує абонента у телефонній книзі. Тип повинен містити: номер телефону, ПІБ абонента, місто, в якому він проживає. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим прізвищем (не ім'ям) виводить на екран всю інформацію про відповідного абонента з масиву даних.
5) Написати функцію, яка за заданим текстовим символом виводить на екран прізвища всіх абонентів, що проживають у містах, назви яких починаються з цього символу.
6) Написати функцію, яка самостійно аналізує масив даних і виводить на екран дані абонентів, які підключені до одного і того самого мобільного оператора (за першими 3 цифрами номеру телефону). Якщо таких груп абонентів декілька, мають бути виведені дані всіх груп.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та `signed int`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 13

1) Визначити перелік, в якому зберігаються різні типи операцій, які може виконувати робот-пилосос. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в

переліку. У випадку співпадіння функція має виводити на консоль назву відповідної операції, інакше – виводити повідомлення, що операція невідома.

2) Визначити структурований тип даних, який описує респондента у поштовій програмі. Тип повинен містити: електронну адресу, псевдонім респондента та його дату народження. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданою електронною адресою виводить на екран всю інформацію про відповідного респондента з масиву даних.
5) Написати функцію, яка за заданим текстовим символом виводить на екран псевдоніми всіх респондентів, імена яких починаються з цього символу.
6) Написати функцію, яка аналізує масив даних і виводить на екран список респондентів, упорядкований за датою народження. Якщо дата народження однакова, упорядковування має відбуватися за алфавітом.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `double` та `unsigned int`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 14

1) Визначити перелік, в якому зберігаються різні типи компонент автомобіля. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного компонента, інакше – виводити повідомлення, що компонент невідомий.
2) Визначити структурований тип даних, який описує абонента у телефонній книзі. Тип повинен містити: номер телефону, ПІБ абонента, та дату останнього дзвінка. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим прізвищем виводить на екран всю інформацію про відповідного абонента з масиву даних.
5) Написати функцію, яка за заданим набором із трьох цифр виводить на екран телефони всіх абонентів, що містять вказану комбінацію цифр.
6) Написати функцію, яка за двома вказаними датами виводить на екран ПІБ абонентів, чий останній дзвінок був здійснений в проміжок між цими

датами.

7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та `double`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об’єднання.

Варіант 15

1) Визначити перелік, в якому зберігаються різні види фруктів. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного фрукту, інакше – виводити повідомлення, що фрукт не знайдено.
2) Визначити структурований тип даних, який описує записи про книги в бібліотеці. Тип повинен містити: назву книги, ПІБ автора, рік видання, кількість сторінок. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданою назвою книги виводить на екран всю інформацію про книгу з масиву даних.
5) Написати функцію, яка за заданим роком видання виводить на екран всі книги, видані після зазначеного року.
6) Написати функцію, яка сортує та виводить на екран всі записи в алфавітному порядку прізвищ авторів.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та `unsigned short`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об’єднання.

Варіант 16

1) Визначити перелік, в якому зберігаються різні типи будівель. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної будівлі, інакше – виводити повідомлення, що будівля невідома.
2) Визначити структурований тип даних, який описує записи про книги в бібліотеці. Тип повинен містити: назву книги, ПІБ автора, рік видання, кількість сторінок. Перейменувати цей тип за допомогою `typedef`.

3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим прізвищем автора виводить на екран всі книги цього автора з масиву даних.
5) Написати функцію, яка за заданої кількістю сторінок виводить на екран всі книги, кількість сторінок в яких більша від зазначеної.
6) Написати функцію, яка сортує та виводить на екран всі записи в порядку збільшення року видання.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та масиву з 4 елементів типу `char`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 17

1) Визначити перелік, в якому зберігаються різні типи наукового обладнання (приладів). Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного обладнання (приладу), інакше – виводити повідомлення, що прилад не знайдено.
2) Визначити структурований тип даних, який описує записи про книги в бібліотеці. Тип повинен містити: назву книги, ПІБ автора, рік видання, кількість сторінок. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 3 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданою кількістю сторінок виводить на екран інформацію про всі книги, що містять саме таку кількість сторінок.
5) Написати функцію, яка за заданим символом виводить на екран всі назви книжок, що починаються з цього символу.
6) Написати функцію, яка сортує та виводить на екран всі записи в порядку збільшення кількості сторінок.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float` та масиву з 2 елементів типу `short`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 18

1) Визначити перелік, в якому зберігаються різні види овочів. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідного овочу, інакше – виводити повідомлення, що овоч невідомий.
2) Визначити структурований тип даних, який описує записи про автомобілі в базі даних. Тип повинен містити: назву (марку) автомобіля, його номер, рік випуску, прізвище власника. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 5 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим прізвищем виводить на екран інформацію про всі автомобілі, які належать цій людині.
5) Написати функцію, яка за заданою маркою виводить на екран інформацію про всі автомобілі цієї марки.
6) Написати функцію, яка сортує та виводить на екран всі записи в порядку збільшення року випуску.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об’єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `float`, `int` та масиву з 4 елементів типу `unsigned char`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об’єднання.

Варіант 19

1) Визначити перелік, в якому зберігаються різні типи доступних комп’ютерних програм. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідної програми, інакше – виводити повідомлення, що програма невідома.
2) Визначити структурований тип даних, який описує записи про автомобілі в базі даних. Тип повинен містити: назву (марку) автомобіля, його номер, рік випуску, прізвище власника. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 5 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим роком випуску виводить на екран інформацію про всі автомобілі, виготовлені цього року.
5) Написати функцію, яка за заданим символом виводить на екран

інформацію про всі автомобілі, назви яких починаються з цього символу.

6) Написати функцію, яка сортує за алфавітом та виводить на екран всі записи, пов'язані із прізвищем власника.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `double` та масиву з 8 елементів типу `unsigned char`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Варіант 20

1) Визначити перелік, в якому зберігаються різні види ліків. Реалізувати функцію, яка порівнює значення переданого їй параметра з символьними константами, визначеними в переліку. У випадку співпадіння функція має виводити на консоль назву відповідних ліків, інакше – виводити повідомлення, що ліки невідомі.
2) Визначити структурований тип даних, який описує записи про автомобілі в базі даних. Тип повинен містити: назву (марку) автомобіля, його номер, рік випуску, прізвище власника. Перейменувати цей тип за допомогою `typedef`.
3) Створити масив зі змінних вказаного типу (не менше 5 елементів) та заповнити його даними. Заповнення даними має відбуватися в коді самої програми, а не через введення даних з консолі.
4) Написати функцію, яка за заданим номером виводить на екран всю інформацію про автомобіль за цим номером.
5) Написати функцію, яка за заданим символом виводить на екран інформацію про всі автомобілі, прізвища власників яких починаються з цього символу.
6) Написати функцію, яка сортує та виводить на екран всі записи в порядку збільшення номера автомобіля.
7) Самостійно підібрати дані, які б демонстрували можливості функцій, вказаних у п. 4-6.
8) Використовуючи об'єднання та бітові поля, реалізувати структуру для роботи з даними, що зберігаються в полях типу `double`, та масиву з 2 елементів типу `unsigned int`. Залежно від значення «прапорця», довжиною 1 біт, мають використовуватись різні поля всередині об'єднання.

Контрольні запитання

1. У чому відмінність між стандартними (вбудованими в мову C/C++) та нестандартними типами? Чи є структуровані типи нестандартними? Які

структуровані типи ви знаєте?

2. Для чого використовується службове слово `typedef`? Наведіть приклади використання `typedef`.
3. Опишіть синтаксис та особливості реалізації структурованих типів даних за допомогою `enum`. Для чого зазвичай застосовується цей тип даних? Наведіть приклади.
4. Що таке операція дозволу видимості `::`? Наведіть приклади її використання для доступу до елементів переліку.
5. Опишіть синтаксис та особливості реалізації структурованих типів даних за допомогою `struct`. Для чого типово застосовується цей тип даних? Наведіть приклади.
6. Якими способами можна звернутися до конкретного поля екземпляру структури (`struct`)? Чи можна змінювати вміст полів структури? Наведіть приклади.
7. Опишіть синтаксис та особливості реалізації структурованих типів даних за допомогою `union`. Для чого типово застосовується цей тип даних? Наведіть приклади.
8. Що таке операція «крапка» `.` та операція «стрілка» `->`? В яких випадках і для яких структурованих типів даних вони застосовуються? Наведіть приклади.
9. Чим структури відрізняються від масивів? Що спільного між цими типами даних? Чи може розмір структури бути більшим за розмір масиву з елементами іншого типу? Того ж самого типу? Якщо відповідь позитивна, наведіть приклад, що підтверджує відповідь.
10. Якщо `T` – це тип деякої структури, чи може вона містити в собі поле типу `T`? Поле типу `T*`? Масив елементів типу `T`? Масив вказівників типу `T*`?
11. Поясніть як за допомогою об'єднань (`union`) можна виконати перетворення даних з однієї форми представлення в іншу? Наведіть приклад коду, що виконує таку операцію.
12. Що таке бітове поле? Чим бітові поля відрізняються від структур? Які обмеження накладаються на поля всередині бітового поля? Наведіть приклади використання бітових полів.
13. Чи можна визначити адресу поля всередині бітового поля? Відповідь пояснити.
14. Чи може функція мати параметри у вигляді структур, об'єднань або бітових полів? Чи може вона повертати результат у вигляді структури, об'єднання або бітового поля? Навести приклади таких функцій.
15. Чому функції для роботи зі структурами, об'єднаннями та бітовими полями часто працюють не з самими екземплярами цих структурованих типів, а з посиланнями або вказівниками на відповідні екземпляри? Наведіть приклади.
16. Чим відрізняються можливості функції, яка приймає параметр у вигляді або звичайного, або сталого посилання на екземпляр структури? Чи може

така функція зчитувати дані полів структури? За яких умов така функція може модифікувати дані полів структури? Наведіть приклад.

Лабораторна робота № 8.
Об'єктно-орієнтоване програмування. Класи C++.
Динамічні структури даних

Мета роботи:

1) Ознайомитись з принципами об'єктно-орієнтованого програмування.
2) Ознайомитись з поняттям класу (`class`) у мові C++. Навчитись створювати та використовувати класи C++ у власних програмах.
3) Ознайомитись з принципами роботи найбільш поширених динамічних структур даних – динамічних масивів та списків. Навчитись створювати та використовувати динамічні масиви та списки в програмах на C++.

КОНЦЕПЦІЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Об'єктно-орієнтоване програмування (ООП) – концепція (парадигма) програмування, згідно якої програма розглядається як сукупність *об'єктів*, що взаємодіють між собою.

Для реалізації ООП у мові C++ використовуються структуровані типи даних, до складу яких входять як члени-дані (поля), так і *члени-функції*, які називаються ***методами***. Найбільш повно принципи ООП реалізуються за допомогою *класів*. ***Клас*** (`class`) – це структурований тип даних C++, що може містити в собі члени-дані (поля) та члени-функції (методи) і забезпечує різні специфікації доступу до членів класу. Окрім класів, для об'єктно-орієнтованого програмування на C++ можна використовувати структури, об'єднання та бітові поля, які з точки зору ООП розглядаються як аналоги класів[1].

Згідно концепції ООП ***об'єктом*** можна назвати будь-який компонент програми, що має визначені властивості, поведінку та стан. Таким чином, кожен екземпляр деякого класу (структури, об'єднання, бітового поля) є ***об'єктом***. Кожен об'єкт має унікальний, характерний саме для нього, набір даних-членів, однак функції-члени (методи) для всіх об'єктів одного типу є спільними. Це дозволяє дані різних об'єктів зберігати окремо і незалежно, але при цьому мати єдиний код (алгоритми) для роботи з даними усіх однотипних об'єктів.

Структура програм, що використовують ООП, дещо відрізняється від структури програм, створених за допомогою методів функціонального програмування (див. лабораторну роботу № 3). Малі за розміром, досить прості

[1] Далі будемо переважно обговорювати реалізацію ООП на основі класів. Однак відповідні принципи ООП майже без змін можуть бути застосовані і для структур, об'єднань та бітових полів.

програми часто пишуть без використання ООП. Однак для програм середньої і великої складності використання ООП, навпаки, дає значні переваги: дозволяє помітно пришвидшити написання програм, зробити програми більш модульними, надійними, полегшити модернізацію коду програми, зробити код програм більш універсальним, гнучким тощо. Тому переважна більшість сучасних професійних програмних продуктів створюються саме за допомогою методів ООП.

ООП базується на трьох основних принципах – це інкапсуляція, успадкування (наслідування) та поліморфізм.

Інкапсуляція – це приховування від зовнішнього коду (глобальних і статичних функцій, методів інших об'єктів) певних деталей реалізації об'єкта. Інкапсуляція, перш за все, «захищає» внутрішній стан об'єкта, не дозволяє зовнішньому коду безпосередньо змінювати цей стан. Це дозволяє об'єкту самому слідкувати за своїм внутрішнім станом, уникати неконтрольованого доступу до даних-членів, допомагає усунути помилки, пов'язані з ініціалізацією або переініціалізацією об'єкта тощо.

На практиці інкапсуляція реалізується за допомогою ***специфікаторів*** або ***ключів доступу***, які приписуються кожному елементу класу (структури, об'єднання або бітового поля). У C++ ці специфікатори позначаються службовими словами `public`, `protected` та `private`. Специфікатор доступу `public` позначає ***відкриті*** елементи, які є завжди доступними для будь-якого коду. Специфікатор доступу `protected` застосовується для ***захищених*** елементів, які мають бути доступні тільки для коду даного класу (структури, об'єднання або бітового поля), або похідних від нього класів[1]. Нарешті, останній специфікатор доступу `private` робить елемент класу ***закритим*** – тобто доступним тільки для коду даного класу (і недоступним для зовнішнього коду та коду класів-нащадків).

Для того щоб зовнішній код міг взаємодіяти з об'єктом програміст має передбачити для об'єкта один або декілька відкритих методів. Цей набір методів називається ***інтерфейсом*** класу (структури, об'єднання, бітового поля). Зовнішній код може викликати ці відкриті методи і за рахунок цього впливати на стан об'єкта або отримувати інформацію про його стан. Важливо підкреслити, що зовнішній код не безпосередньо звертається до стану об'єкта (наприклад, змінює/зчитує вміст полів всередині об'єкта), а робить це через код інтерфейсу, який «знає» внутрішню структуру об'єкта і вміє з нею працювати. Більше того, якщо програміст змінить внутрішню організацію об'єкта (формат збереження даних, імена полів тощо), але інтерфейс класу – набір відкритих методів класу – залишить незмінним, для зовнішнього коду зроблені в об'єкті зміни будуть непомітними. Це дозволяє за необхідності змінювати внутрішню будову об'єктів, залишаючи незмінними їх інтерфейси, що в цілому ніяк не позначається на працездатності інших частин програми.

---

[1] Похідний клас – це клас-нащадок, який утворюється в процесі успадкування з базового класу. Детальніше про це див. далі.

Успадкування (наслідування) – це можливість набуття властивостей і методів одного класу іншим. При цьому виділяють ***базовий*** або ***батьківський клас*** і ***клас-нащадок*** або ***похідний клас***. Успадкування може бути ***простим*** (одинарним, звичайним) або ***складним*** (***множинним***). При простому успадкуванні клас-нащадок успадковує властивості та методи тільки одного базового класу. За множинного успадкування клас-нащадок може успадкувати властивості та методи кількох базових класів.

Зазначимо, що будь-який клас-нащадок окрім успадкованих елементів (полів і методів) базового класу (або базових класів), може містити довільну кількість додаткових полів та методів, специфічних саме для нього. Завдяки цим додатковим специфічним елементам клас-нащадок буде принципово відрізнятись від батьківського класу (батьківських класів).

Поліморфізм – це можливість виконання різних дій з об'єктами різного типу за допомогою одного і того ж самого фрагменту коду, інтерпретація (суть) якого визначається *типом* об'єкта.

Поліморфізм є доволі складною технологією, схожою на технологію перевантаження функцій, яка розглядалась у лабораторній роботі № 3. У програмах на C++ поліморфізм дозволяє перевантажувати методи певного класу, так само як перевантажуються звичайні функції. Але окрім цього, поліморфізм у C++ дозволяє перевантажувати методи *різних* класів, зокрема, й класів пов'язаних успадкуванням, – тобто задавати для всіх цих методів *різних* класів однакове ім'я. Отже, фрагмент коду, в якому викликається один з таких методів, може виглядати *однаково* для *різних* класів, проте результат виконання коду буде *різним* для об'єктів *різного* класу. Поліморфізм буде більш детально обговорюватись пізніше. Поки що лише зазначимо, що у мові C++ поліморфізм тісно пов'язаний з механізмом ***віртуальних функцій*** або ***механізмом пізнього*** (***динамічного***) ***зв'язування*** коду.

У програмах на C++ поліморфізм дозволяє писати єдиний однаковий код для роботи з різними об'єктами, що робить програмний код більш уніфікованим і водночас більш гнучким. Крім того, поліморфізм лежить в основі технології перевантаження операторів, і застосовується при перевизначенні («перекритті») всередині класу-нащадка методів, успадкованих від базового класу, тощо.

Класи C++

Оголошення класів

Клас (`class`) – це структурований тип даних C++, що може містити в собі члени-дані (поля) та члени-функції (методи) і забезпечує різні специфікації доступу до членів класу. Оголошення класів дещо нагадує оголошення структур (див. лабораторну роботу № 7). Проте, на відміну від структур мови

С, класи можуть включати у себе не тільки поля (члени-дані), але й методи (нагадаємо, що у С++ структури, об'єднання та бітові поля також можуть мати елементи-функції і розглядаються як аналоги класів):

```
class Ім'я_Класу
{
Специфікатор_доступу_П1:
     Тип1 Ім'я1;
Специфікатор_доступу_П2:
     Тип2 Ім'я2;
     //...
Специфікатор_доступу_ПN:
     ТипN Ім'яN;

Специфікатор_доступу_М1:
     Метод_1( /* ... */ );
Специфікатор_доступу_М2:
     Метод_2( /* ... */ );
     // ...
Специфікатор_доступу_МX:
     Метод_X( /* ... */ );
};
```

`Ім'я_ Класу` є унікальним ідентифікатором, який є ім'ям нового типу, пов'язаного із класом.

Елементи `Тип1 Ім'я1`, `Тип2 Ім'я2`, … `ТипN Ім'яN` визначають поля класу. Кожне поле задається своїм типом (`Тип1`, `Тип2`, …) та ім'ям (`Ім'я1`, `Ім'я2`, …).

Окрім елементів-полів, клас може мати елементи-функції – ***методи***, які умовно позначені як `Метод_1( /* ... */ )`, `Метод_2( /* ... */ )`, …, `Метод_X( /* ... */ )`. Кожен метод, за виключенням *конструкторів* та *деструктора*, які будуть розглядатись пізніше, має стандартний для функції формат запису. Цей формат запису включає в себе тип результату, який повертає метод, ім'я методу, перелік його аргументів. Також, подібно до звичайних функцій, методи класу можуть мати додаткові модифікатори, наприклад, модифікатори `inline`, `static` та деякі інші.

Поля та методи можна оголошувати у довільному порядку (навіть чергуючи їх між собою). На практиці, однак, виявляється доцільно дотримуватись певного єдиного стилю запису. Досить часто використовується стиль запису наведений вище, коли спочатку оголошуються всі поля і лише потім всі методи, або, навпаки, зворотній спосіб запису – спочатку оголошуються всі методи і лише потім поля.

Імена полів та методів мають бути дозволеними ідентифікаторами, унікальними *лише у межах даного класу*. Різні класи можуть мати поля/методи з

однаковими іменами та/або типами.

Типи, які використовуються для запису полів та методів класу, мають бути визначені (відомі) на момент оголошення класу. Це можуть бути як вбудовані типи, так і будь-які інші типи, введені програмістом, включаючи вказівники, посилання та масиви даних довільного типу. Однак, так само як для структур, заборонено оголошувати і створювати класи, елементами яких є самі ці класи. Проте дозволяється оголошувати класи, в яких присутні вказівники на ці класи.

При оголошенні класу для кожного його елементу може бути вказаний один з трьох дозволених специфікаторів доступу: `public`, `protected` або `private`. Ці специфікатори доступу для полів позначені як `Специфікатор_доступу_П1`, `Специфікатор_доступу_П2`, …, `Специфікатор_доступу_ПN`, а для методів як `Специфікатор_доступу_М1`, `Специфікатор_доступу_М2`, …, `Специфікатор_доступу_МХ`.

Нагадаємо, що специфікатор доступу `public` позначає ***відкриті*** елементи, які є завжди доступними для будь-якого коду. Специфікатор доступу `protected` позначає ***захищені*** елементи, які будуть доступні тільки для коду даного класу (структури, об'єднання або бітового поля), або похідних від нього класів. Нарешті, специфікатор доступу `private` робить елемент класу ***закритим*** – тобто доступним тільки для коду даного класу.

Кожен специфікатор доступу впливає на всі елементи класу (як поля, так і методи), які йдуть в оголошенні класу після цього специфікатора. Таким чином, специфікатор може керувати доступом до довільної кількості елементів, які йдуть після нього. Саме тому, оголошення класу часто представляють у більш уніфікованій формі запису:

```
class Ім'я_Класу
{
public:
     Тип1 Відкр_Ім'я1;
     // Оголошення інших відкритих полів...

protected:
     Тип2 Захищ_Ім'я2;
     // Оголошення інших захищених полів...

private:
     Тип3 Закр_Ім'я3;
     // Оголошення інших закритих полів...

public:
     Метод_1( /* ... */ );
```

```
    // Оголошення інших відкритих методів...

protected:
    Метод_2( /* ... */ );
    // Оголошення інших захищених методів...

private:
    Метод_3( /* ... */ );
    // Оголошення інших закритих методів...
};
```

Або використовують ще більш компактну форму запису:

```
class Ім'я_Класу
{
public:
    // Оголошення відкритих полів і методів...

protected:
    // Оголошення захищених полів і методів...

private:
    // Оголошення закритих полів і методів...
};
```

Якщо специфікатор доступу для елемента класу не зазначено, за замовчуванням використовується специфікатор доступу `private`. Отже, за замовчуванням всі елементи класу вважаються *закритими*! В той же час у структурах, об'єднаннях та бітових полях за замовчуванням використовується специфікатор доступу `public`. Тобто в мові C++ елементи структур, об'єднань та бітових полів за замовчуванням вважаються *відкритими*, так само як і в мові C. Це, однак, означає, що для відкритих елементів вказаних структурованих типів порушується принцип інкапсуляції.

Якщо клас не має жодного елемента, він все одно буде вважатись деяким новим унікальним типом даних з ім'ям `Ім'я_Класу`.

Раніше, у лабораторній роботі № 7, була введена структура `Student`, яка може зберігати інформацію про студента університету. Розглянемо аналог цієї структури – клас `CStudent`, який містить тільки елементи-поля:

```
class CStudent
{
public:
    // Прізвище Ім'я По батькові.
    char   Name[64];
```

```
protected:
    // Дата народження: Д-М-РР.
    unsigned DoB;
    // Курс.
    int    YearOfStudy;
    // Група.
    char   Group[16];

private:
    // Оцінка = середній бал за сесію.
    double Grade;
};
```

У даному прикладі символьний масив `Name`, який зберігає інформацію про прізвище, ім'я та по батькові студента, є відкритим елементом, який має специфікатор доступу `public`. Дата народження студента, номер курсу та назва групи є захищеними (`protected`) елементами. Безпосередній доступ зовнішнього коду до цих елементів заборонений. Останній елемент `Grade`, що зберігає інформацію про оцінку студента, є закритим елементом (`private`). Він буде доступний тільки методам, оголошеним всередині класу (яких поки що немає).

Іншим прикладом оголошення класу є клас `CComplex`, який є аналогом структури `Complex`, введеної у лабораторній роботі № 7 для збереження даних комплексного числа:

```
class CComplex
{
    double re, im;
};
```

Так само, як і в структурі `Complex`, поля `re` та `im` класу зберігають інформацію про дійсну та уявну частини комплексного числа. Проте, на відміну від полів структури `Complex`, поля `re` та `im` класу `CComplex` є закритими, оскільки за замовчуванням для елементів класів використовується специфікатор доступу `private`. Отже, доступ зовнішнього коду до полів `re` та `im` класу `CComplex` є забороненим. Тому, щоб зберігати, зчитувати та модифікувати дані такого класу, до нього необхідно додати відкриті методи – тобто реалізувати *інтерфейс* для цього класу. Приклад того, як це можна зробити, буде розглянуто далі.

Операції з об’єктами

Створення об’єкта деякого відомого (оголошеного) класу (структури, об’єднання, бітового поля) відбувається під час оголошення екземпляра (об’єкта) відповідного класу. Це здійснюється за загальними правилами, так само як і для змінних вбудованого типу. Наприклад, фрагмент коду

```
CStudent stud1, stud2;
CComplex c1, c2, c3;
```

створює два *об’єкти* (або *екземпляри*) `stud1` та `stud2` типу `CStudent`, а також три *об’єкти* `c1`, `c2`, `c3` типу `CComplex`.

Подібно структурам у мові С, об’єкти з усіма відкритими полями можна визначати, явно задаючи початкові значення для цих полів (див. лабораторну роботу № 7). Проте такий підхід спричиняє порушення принципу інкапсуляції даних, тому його використовують не дуже часто. Більш поширеною методикою є використання різних видів конструкторів для ініціалізації об’єктів класу (про це див. далі).

Операції над екземплярами класів виконуються за тими ж правилами, що й операції над екземплярами структур мови С (див. лабораторну роботу № 7). Як правило, компілятор автоматично реалізує для об’єктів тільки операцію присвоювання `=` і операцію визначення адреси `&`. За необхідності, можна навчити програму виконувати й інші операції над об’єктами класу, використовуючи технологію перевантаження операторів (див. лабораторну роботу № 9).

Операція присвоювання `=` для двох однотипних об’єктів, зазвичай, реалізується як операція побайтового копіювання вмісту одного об’єкту в інший. Наприклад, фрагмент коду `stud2 = stud1;` приведе до копіювання даних з екземпляра класу `stud1` типу `CStudent` в об’єкт `stud2` того ж класу. За необхідності програміст може перевизначити цю операцію, використовуючи технологію перевантаження операторів (див. лабораторну роботу № 9).

Операція визначення адреси `&` реалізується за загальними правилами, так само як для структур С та змінних вбудованих типів. Однак, за необхідності, цю операцію можна перевантажити (див. лабораторну роботу № 9).

Оголошення та визначення методів

В С++ функції можуть бути *елементами* класів (структур, об’єднань та бітових полів). Такі функції прийнято називати ***методами***.

Оголошення методів відбувається подібно оголошенню звичайних (глобальних) функцій (див. лабораторну роботу № 3). Єдиною відмінністю є те, що звичайні функції можуть оголошуватись у будь-якому місці програми, а методи мають оголошуватись всередині того класу (структури, об’єднання,

бітового поля), якому вони належать.

Оголошення методів синтаксично не відрізняється від оголошення функцій (див. лабораторну роботу № 3). Для кожного з методів задається тип результату, який він повертає, унікальне ім'я методу, його параметри. Подібно до звичайних функцій, методи можуть мати додаткові модифікатори (`inline`, `static` тощо).

Відмінність між звичайними функціями і методами проявляється у синтаксисі їх визначення. При визначенні методу ззовні від оголошення класу (структури, об'єднання, бітового поля) `Ім'я_Класу` необхідно обов'язково вказувати додатковий «префікс» `Ім'я_Класу::` перед іменем кожного метода. Цей «префікс» інформує компілятор, що відповідна функція не є глобальною, а є *методом*, асоційованим з вказаним типом `Ім'я_Класу`.

Якщо ж тіло метода визначається безпосередньо всередині класу, записувати «префікс» `Ім'я_Класу::` взагалі непотрібно. До того ж, методи, визначені всередині оголошення класу, як правило, розглядаються як вбудовані (залежно від налаштувань компілятора), навіть якщо вони записані без використання службового слова `inline`[1].

За замовчуванням методи мають повний доступ до елементів класу (структури, об'єднання, бітового поля) і можуть вільно зчитувати та записувати дані полів. Але іноді виникає необхідність реалізувати метод, який має *тільки зчитувати* дані полів, однак не повинен їх змінювати. Щоб зробити це, метод слід оголосити зі службовим словом `const`, записаним після круглих дужок, в яких зазначаються аргументи. Введений таким чином модифікатор метода `const` інформує компілятор про те, що метод не може змінювати вміст пов'язаного з ним об'єкта[2].

У наступному лістингу 8.1 всередині класу `CComplex` визначаються три відкритих метода: `Arg()`, `Re()` та `Re( double _re )`:

Лістинг 8.1:

```
 1: #include <stdio.h>
 2: #include <math.h>
 3:
 4: class CComplex
 5: {
 6:     double re, im;
 7:
 8: public:
 9:     double Arg() const;
```

[1] Ці методи будуть реалізовані як вбудовані, тільки якщо вони задовольняють відповідним вимогам до вбудованих функцій.

[2] Якщо метод з модифікатором `const` змінює вміст об'єкта, відповідний фрагмент тексту програми компілюватись не буде. Коли компілятор зустрічає такий некоректно реалізований метод, він виводить повідомлення про помилку і переривает процес компіляції.

```
10:      double Re () const { return re; }
11:      void   Re( double _re ) { re = _re; }
12: };
13:
14: double CComplex::Arg() const
15: {
16:      return atan2(im, re);
17: }
```

Метод `Arg()` оголошується в рядку 9 і визначається ззовні від оголошення класу (рядки 14-17), що супроводжується використанням «префіксу» `CComplex::`. Цей метод розраховує аргумент (фазу) комплексного числа. Він не може змінювати вміст об'єкта класу `CComplex`, на що вказує модифікатор методу `const` (див. рядки 9, 14).

Методи `Re()` та `Re( double _re )` є перевантаженими. Метод `Re()`, оголошений і одночасно визначений в рядку 10, повертає значення дійсної частини комплексного числа. Він не може змінювати вміст об'єкта класу `CComplex`. Інший метод, `Re( double _re )`, навпаки, не повинен мати модифікатор `const`, оскільки переініціалізує дійсну частину комплексного числа (рядок 11). Разом обидва методи `Re()` та `Re( double _re )` дозволяють зчитувати та записувати вміст дійсної частини комплексного числа (поля `re` в класі `CComplex`).

Методи та поля класу (структури, об'єднання, бітового поля) задаються в ***області дії*** класу, унікальній для кожного окремого класу[1]. Це дозволяє всередині методів класу звертатись до елементів цього класу (полів та методів) за «спрощеною процедурою» – безпосередньо через їх імена, вказані при оголошенні класу (структури, об'єднання, бітового поля). При цьому код методів автоматично отримує доступ до полів саме того об'єкта (екземпляра класу), для якого викликався метод. Завдяки цій особливості мови C++ запис тексту програми на C++ помітно спрощується. Так, при визначенні методів у лістингу 8.1 не потрібно явно вказувати з полями якого конкретного об'єкта мають працювати методи. За правилами C++ ці методи будуть завжди працювати з полями лише того об'єкта, для якого вони викликаються, тому в коді методів достатньо оперувати тільки іменами полів.

Конструктори та деструктор

Окрім «звичайних» методів, яких може бути довільна кількість і які

[1] У мові програмування C існують різні області дії змінних та функцій: блок коду, функція, .c/.cpp файл, вся програма (зокрема, з кількох .c/.cpp файлів). У мові C++ до них додається ще одна область дії: клас (структура, об'єднання, бітове поле).

можуть виконувати різні задачі і мати довільні дозволені імена, з кожним класом (структурою, об'єднанням, бітовим полем) пов'язують ще спеціальні методи – *конструктор* та *деструктор*.

Конструктор – це спеціальний метод, який має *те ж саме ім'я*, що й тип об'єкта, і автоматично викликається відразу ж після створення об'єкта. Завдяки цій особливості, конструктори, зазвичай, використовують для ініціалізації об'єктів.

Конструктори є незвичайними методами. Вони мають наперед визначене ім'я у вигляді `Ім'я_Класу()`, де `Ім'я_Класу` – це ім'я типу об'єкта. Наприклад, для класу `CComplex`, розглянутого вище, конструктор має ім'я `CComplex()`. Конструктор також *не може* повертати будь-яких значень, тому для конструктора не вказується тип результату, який він повертає.

Конструкторів може бути декілька (це реалізується за допомогою технології перевантаження функцій).

Конструктор, який не має аргументів, називається ***конструктором за замовчуванням***. Він викликається для об'єктів, для яких не вказуються дані ініціалізації. Наприклад, коли за допомогою рядка коду `CComplex x;` створюється об'єкт `x` класу `CComplex`, для ініціалізації цього об'єкта автоматично викликається конструктор за замовчуванням `CComplex()`.

Іншим видом конструктора, який часто вживається на практиці, є ***конструктор копії***, який має формат запису `Ім'я_Класу( const Ім'я_Класу& )`. Цей конструктор ініціалізує об'єкт даними іншого однотипного з ним об'єкта, заданого через стале посилання `const Ім'я_Класу&`.

Особливим випадком конструктора є т. зв. ***пустий конструктор***, який не виконує жодних дій. Наприклад, для класу `CComplex` конструктор за замовчуванням може бути реалізований як пустий: `CComplex(){}`. Такий конструктор буде викликатись, але ніякої корисної дії – тобто ініціалізацію полів `re` та `im` класу – він виконувати не буде.

У кожного класу (структури, об'єднання, бітового поля) може бути *тільки один* ***деструктор*** – спеціальний метод, який викликається перед знищенням об'єкта. Зазвичай, деструктор використовуються для звільнення ресурсів, пов'язаних з об'єктом, наприклад, звільнення виділених блоків динамічної пам'яті, закриття файлів тощо.

Деструктор має наперед визначене ім'я – це ім'я типу, перед яким записується знак тильди `~`, тобто його ім'я `~Ім'я_Класу`. Наприклад, для класу `CComplex` деструктор має ім'я `~CComplex()`.

Так само як і конструктор, деструктор не повертає жодних значень, а тому тип результату для нього не вказується.

Деструктор може бути пустим. У цьому випадку його тіло, обмежене дужками `{}`, не містить жодного оператора.

У лістингу 8.2 наведено приклад реалізації кількох конструкторів і пустого деструктора для раніше розглянутого класу `CComplex`:

Лістинг 8.2:

```
 1: #include <stdio.h>
 2: #include <math.h>
 3:
 4: class CComplex
 5: {
 6:     double re, im;
 7:
 8: public:
 9: // Конструктори.
10:     CComplex(){ re = 0.0; im = 0.0; }
11:     CComplex( double _re )
12:     {
13:         re = _re;
14:         im = 0.0;
15:     }
16:     CComplex( double _re, double _im );
17:     CComplex( const CComplex& r );
18:     // Деструктор.
19:     ~CComplex(){}
20: };
21:
22: CComplex::CComplex( double _re, double _im )
23: {
24:     re = _re;
25:     im = _im;
26: }
27:
28: CComplex::CComplex( const CComplex& r )
29: {
30:     re = r.re;
31:     im = r.im;
32: }
33:
34: int main( void )
35: {
36:     // Конструктор за замовчуванням:
37:     CComplex a;             // re = im = 0.0
38:     // Конструктор CComplex( double ):
39:     CComplex b(1.0);        // re = 1.0; im = 0.0
40:     // Конструктор CComplex( double, double ):
41:     CComplex c(-3.0, 5.0);// re = -3.0; im = 5.0
42:     // Конструктор CComplex( const CComplex& ):
43:     CComplex d(c);          // re = -3.0; im = 5.0
```

```
44:
45:       return 0;
46:  }
```

У рядку 10 визначається і одночасно оголошується конструктор за замовчуванням `CComplex()`, який ініціалізує поля `re` та `im` класу значенням `0.0`.

Окрім цього конструктора, у лістингу 8.2 вводяться ще 3 конструктора.

Конструктор `CComplex( double _re )` (рядки 11-15) ініціалізує дійсну частину `re` комплексного числа за допомогою переданого параметра `_re`, а уявну частину `im` комплексного числа встановлює у значення нуль.

Конструктор `CComplex( double _re, double _im )` дозволяє явно задавати для об'єкта класу `CComplex` дійсну та уявну частини комплексного числа. Він оголошений у рядку 16 і визначається у рядках 22-26 з використанням «префіксу» `CComplex::`.

У рядку 17 оголошується конструктор копії `CComplex( const CComplex& r )`, який далі визначається в рядках 28-32.

У рядку 19 оголошується і визначається пустий деструктор `~CComplex()`.

Фрагмент коду, в якому використовуються різні конструктори, наведений в рядках 37-43. У рядку 37 оператор `CComplex a;` записано без використання будь-яких значень ініціалізації для об'єкта `a`, тому ініціалізація об'єкта `a` відбувається конструктором за замовчуванням, який встановлює поля `re` та `im` об'єкту `a` в нульові значення.

У наступному операторі `CComplex b(1.0);` вже задається значення типу `double`. Отже, для ініціалізації об'єкта `b` викликається конструктор `CComplex( double _re )`.

Рядок `CComplex c(-3.0, 5.0);` передбачає ініціалізацію об'єкта `c` за допомогою конструктора `CComplex( double _re, double _im )`, який приймає два параметри типу `double`.

Нарешті у рядку 43 – `CComplex d(c);` – викликається конструктор копії `CComplex( const CComplex& r )`, який ініціалізує об'єкт `d` вмістом об'єкта `c`.

Наведений вище код для конструкторів у лістингу 8.2 записано у традиційній формі. У мові C++, однак, дозволяється інша, альтернативна форма запису конструкторів за допомогою т. зв. ***списків ініціалізації***. Згідно такого підходу, будь-яке поле розглядається як деякий об'єкт, який можна ініціалізувати, вказуючи в круглих дужках після його імені потрібне початкове значення (за аналогією як це робилось у рядках 37-43 лістингу 8.2). Такі блоки для ініціалізації різних полів записуються через кому, формуючи список ініціалізації, який починається зі спеціального символу `:`. Сформований список ініціалізації записується перед тілом конструктора, обмеженим `{}`, яке за цієї формі запису часто виявляється пустим. Нижче наведено приклад запису

конструкторів класу `CComplex` за допомогою списків ініціалізації:

Лістинг 8.3:

```
 1: class CComplex
 2: {
 3:     double re, im;
 4:
 5: public:
 6: // Конструктори.
 7:     CComplex() : re(0.0), im(0.0){}
 8:     CComplex( double _re ) :
 9:         re(_re), im(0.0) {}
10:     CComplex( double _re, double _im );
11:     CComplex( const CComplex& r );
12:     // Деструктор.
13:     ~CComplex(){}
14: };
15:
16: CComplex::CComplex( double _re, double _im )
17: : re(_re), im(_im)
18: {
19: }
20:
21: CComplex::CComplex( const CComplex& r )
22: : re(r.re), im(r.im)
23: {
24: }
```

Вказівник this

При виклику кожного нестатичного (без модифікатора `static`) методу, йому приховано передається вказівник `this` на той об'єкт, для якого викликався метод. Завдяки цьому кожен метод завжди «знає» з яким об'єктом він працює і має повний доступ до даних цього об'єкту.

За замовчуванням робота методів з вказівником `this` відбувається приховано від програміста. За реалізацію цієї роботи відповідає компілятор – він автоматично створює код для передачі та використання вказівника `this` у будь-якому нестатичному методі. Разом з тим, програміст також може використовувати вказівник `this` у тілі нестатичних методів. Приклад того, як це можна реалізувати, наведено нижче:

Лістинг 8.4:

```
 1: #include <stdio.h>
 2: #include <math.h>
 3:
 4: class CComplex
 5: {
 6:     double re, im;
 7:
 8: public:
 9:     double Arg() const;
10:     double Re () const { return re; }
11:     void   Re( double _re ) { this->re = _re; }
12: };
13:
14: double CComplex::Arg() const
15: {
16:     return atan2(this->im, this->re);
17: }
```

В рядку 11, у тілі методу `Re( double _re )`, для зміни вмісту поля `re` використовується оператор `this->re = _re;`. Звернення до поля `re` об'єкту відбувається не безпосередньо, як у лістингу 8.1, а через операцію доступу «стрілка» `->`, що застосовується до вказіника `this`.

Аналогічним чином у рядку 16 функції `atan2()` передаються значення полів `im` та `re`, доступ до яких отримується як `this->im` та `this->re`.

Доступ до полів класу і виклик його методів

За своїм синтаксисом звернення до полів класу ідентичне зверненню до полів структур, об'єднань та бітових полів. Це відбувається або за допомогою операції доступу «крапка» `.`, або операції доступу через вказівник – операції «стрілка» `->`. Таким чином, щоб звернутись до деякого поля `x` всередині об'єкта `obj` типу `CObject` необхідно використати одну з форм запису:

```
CObject  Obj;
CObject* pObj = &Obj;
// Звернення за допомогою операції "крапка".
Obj.x
// Звернення за допомогою операції "стрілка".
pObj->x
```

Крім того, якщо звернення до поля `x` відбувається всередині нестатичного метода, що належить типу `CObject`, для звернення достатньо лише

записати ім'я відповідного поля – тобто `x`. Саме за таким принципом написаний код у лістингах 8.1-8.3, де звернення до полів `re` та `im` в методах класу `CComplex` відбувалось безпосередньо через імена цих полів `re` та `im`. Також всередині нестатичних методів можна звертатись до полів за допомогою операції `->`, використовуючи прихований вказівник `this`. Приклад коду, що використовує цей формат доступу до полів, наведено в лістингу 8.4.

Виклик методів деякого об'єкта реалізується в два етапи. Спочатку в тексті програми слід *звернутись* до відповідного методу для вказаного об'єкту і потім *викликати* цей метод. Синтаксис звернення до методу ідентичний зверненню до поля об'єкта, а виклик методу реалізується стандартним чином – шляхом запису круглих дужок `()`, у яких вказуються можливі параметри метода. Наприклад, якщо всередині об'єкту `CObject Obj` є метод з іменем `Method()`, щоб викликати його достатньо написати код:

```
CObject  Obj;
CObject* pObj = &Obj;
// Виклик метода за допомогою операції "крапка".
Obj.Method( /* Параметри */ );
// Виклик метода за допомогою операції "стрілка".
pObj->Method( /* Параметри */ );
// Виклик метода всередині іншого метода.
Method( /* Параметри */ );
this->Method( /* Параметри */ );
```

При зверненні до полів об'єкта та виклику його методів слід пам'ятати, що відповідні операції дозволені лише для полів і методів з *належним специфікатором доступу*. Наприклад, у зовнішньому коді, визначеному поза межами області дії класу, дозволяється звертатись лише до відкритих (`public`) полів і методів класу. Спроба звернення зовнішнього коду до захищених (`protected`) або закритих (`private`) елементів класу призведе до помилки компіляції такого коду.

У лістингу 8.5 наведено текст програми, яка працює аналогічно до програми лістингу 7.1. Проте, на відміну від попередньої версії програми, у лістингу 8.5 використовується клас `CComplex`, який містить ряд методів, включаючи конструктори і деструктор.

Лістинг 8.5:

```
 1: #include <stdio.h>
 2: #include <math.h>
 3:
 4: class CComplex
 5: {
 6:     double re, im;
```

```
 7:
 8: public:
 9: // Конструктори.
10:     CComplex(): re(0.0), im(0.0) {}
11:     CComplex( double _re )
12:     {
13:         re = _re;
14:         im = 0.0;
15:     }
16:     CComplex( double _re, double _im )
17:     {
18:         re = _re;
19:         im = _im;
20:     }
21:     CComplex( const CComplex& r )
22:     {
23:         this->re = r.re;
24:         this->im = r.im;
25:     }
26: // Деструктор.
27:     46{}
28:
29: // Методи.
30:     double    Arg() const
31:     {
32:         return atan2(im, re);
33:     }
34:     double     Im() const { return im; }
35:     void  Im( double _im ){ im = _im; }
36:     double     Re() const { return re; }
37:     void  Re( double _re ){ re = _re; }
38:     CComplex& IncBy( const CComplex& z );
39:     double    Module() const
40:     {
41:         return sqrt(re*re + im*im);
42:     }
43:     void      Print() const
44:     {
45:         printf("CComplex( %f , %f )", re, im);
46:     }
47: };
48:
49: CComplex& CComplex::IncBy( const CComplex& z )
50: {
51:     re += z.re;
```

```
52:       im += z.im;
53:       return *this;
54:  }
55:
56:  // Функція для додавання об'єктів CComplex.
57:  CComplex cAdd( const CComplex& a, const CComplex&
58:  b )
59:  {
60:       return CComplex(
61:            a.Re() + b.Re(),
62:            a.Im() + b.Im()
63:       );
64:  }
65:
66:  int main( void )
67:  {
68:       CComplex a(-1.0, 1.0);
69:       CComplex b( 3.0, 7.0);
70:
71:       printf("a == ");
72:       a.Print();
73:       printf("\tb == ");
74:       b.Print();
75:       printf("\n\n");
76:
77:       CComplex c = cAdd(a, b);
78:       printf("c == a + b == ");
79:       c.Print();
80:       printf("\n\n");
81:
82:       printf("c == c + a == ");
83:       c.IncBy(a);
84:       c.Print();
85:       printf("\n\n");
86:
87:       printf("Module of c is %f , argument is
88:  %f\n\n\n", c.Module(), c.Arg());
89:
90:       return 0;
91:  }
```

Програма з лістингу 8.5 використовує різні програмні конструкції, які обговорювались раніше.

У рядках 10-25 записуються кілька конструкторів класу `CComplex`.

Конструктор за замовчуванням `CComplex()` (рядок 10) використовує список ініціалізації `: re(0.0), im(0.0)`. Конструктор копії `CComplex( const CComplex& r )` використовує звернення до полів класу через вказівник `this`. Інші конструктори звертаються до полів об'єкту безпосередньо через імена цих полів.

У рядку 27 визначається пустий деструктор `~CComplex()`.

Відкриті методи класу `CComplex` вводяться в рядках 30-46. Більшість з цих методів, а саме `Arg()`, `Im()`, `Im( double _im )`, `Re()`, `Re( double _re )`, `Module()` та `Print()`, визначаються і одночасно оголошуються всередині оголошення класу. Метод `IncBy( const CComplex& z )` оголошується всередині класу, але визначається ззовні, у рядках 49-54.

Метод `IncBy()` виконує зсув дійсної та уявної частини комплексного числа на величину дійсної та уявної частини іншого комплексного числа `z`. Метод повертає посилання на модифіковане комплексне число як `*this`. Оскільки `this` – це вказівник на об'єкт, тобто вказівник `CComplex*`, розіменування цього вказівника дає сам об'єкт (тип `CComplex`) або посилання на об'єкт (тип `CComplex&`).

У рядках 57-64 визначається глобальна функція `cAdd()`, яка виконує додавання двох комплексних чисел, заданих через сталі посилання `a` та `b`. Для одержання кінцевого результату – нового об'єкта типу `CComplex` – функція використовує конструктор `CComplex( double _re, double _im )`. Виклик цього конструктора створює новий, *безіменний* об'єкт типу `CComplex` із заданими значеннями дійсної та уявної частини, який відразу ж повертається за допомогою оператора `return` у рядку 60. Оскільки поля `re` та `im` класу `CComplex` є закритими, функція `cAdd()` не може звертатись до них безпосередньо, так як це робилось у лістингу 7.1. Тому, щоб отримати значення полів `re` та `im` для об'єктів, заданих через посилання `a` та `b`, функція використовує відкриті методи `Im()`, `Re()`, які повертають значення уявної та дійсної частин для відповідних комплексних чисел.

Використання конструкторів об'єктів демонструється у рядках 68, 69, де задаються об'єкти `a` та `b`, що описують комплексні числа. У рядку 72, 74 вміст цих об'єктів виводиться на консоль за допомогою викликів методу `Print()`.

Рядок `CComplex c = cAdd(a, b);` демонструє виконання операції додавання комплексних чисел за допомогою функції `cAdd()`. Результат цієї операції зберігається в об'єкті `c`. Зазначимо, що можлива й інша форма запису такої операції, а саме: `CComplex c(cAdd(a, b));`, що передбачає використання конструктора копії для ініціалізації об'єкта `c` значенням, що повертає `cAdd()`. Вміст об'єкта `c` виводиться на консоль за допомогою `Print()` (рядок 79).

Далі виконується операція `c += a;` за допомогою виклику методу `c.IncBy(a);`. Після цього вміст об'єкта `c` виводиться на консоль за

допомогою `Print()`.

Останній блок коду (рядки 87, 88) виводить на консоль модуль та аргумент комплексного числа, дані якого зберігаються в об'єкті `c`. Для цього в рядку 88 для об'єкта `c` послідовно викликаються методи `Module()` та `Arg()`.

Результат роботи програми лістингу 8.5 такий самий, як і результат роботи програми лістингу 7.1 – див. рис. 25.

Функції та методи можуть мати аргументи, тип яких дорівнює типу об'єкта, (сталому) вказівнику або посиланню на об'єкт. Також функції та методи можуть повертати величини усіх перелічених вище типів. Частково це ілюструється у лістингу 8.5, де конструктор копії, метод `IncBy()` та функція `cAdd()` мають аргументи типу `const CComplex&`. Метод `IncBy()` повертає результат типу `CComplex&`, а функція `cAdd()` повертає сам об'єкт типу `CComplex`.

Масиви об'єктів класу

Мова C++ дозволяє створювати статичні масиви об'єктів класу. Це відбувається за тими ж самими правилами, що й створення статичних масивів структур у мові C. Наприклад, наступні рядки коду оголошують масив з 30 комплексних чисел типу `CComplex` і масив з 1000 записів типу `CStudent` – тих класів, які були введені раніше:

```
CComplex ca[30];
CStudent students[1000];
```

Для доступу до об'єктів-елементів цих масивів використовуються стандартні правила. Наприклад, щоб змінити дійсну та уявну частину комплексного числа, яке зберігається в елементі масиву з індексом 10, використовується фрагмент коду:

```
ca[10].Re(1.0);
ca[10].Im(-2.0);
```

Звернення `ca[10]` сприймається компілятором як елемент масиву з індексом 10 – тобто об'єкт типу `CComplex`. Далі до цього об'єкту застосовується операція `.`, яка дозволяє викликати відкриті методи `Re(double)` та `Im(double)` для зміни вмісту об'єкту.

Фрагмент коду, що виконує аналогічну роботу, але використовує замість `.` «операцію стрілка» `->`, виглядатиме так:

```
(ca+10)->Re(1.0);
```

```
(ca+10)->Im(-2.0);
```

Мова C++ також дозволяє створювати багатовимірні масиви об'єктів. Правила роботи з такими масивами нічим не відрізняються від правил роботи з багатовимірними масивами для стандартних типів даних (див. лабораторну роботу № 4).

ДИНАМІЧНІ СТРУКТУРИ ДАНИХ

Динамічні структури даних – це структуровані типи (як правило, класи), що дозволяють зберігати довільну кількість елементів довільного типу у динамічній пам'яті. Такі структури даних іноді називають ***динамічними контейнерами*** або просто ***контейнерами***.

Залежно від кількості елементів, які потрібно зберігати, та їх типу, контейнер може автоматично змінювати об'єм динамічної пам'яті, що використовується. За необхідності такий контейнер нібито «стискається» або «розтягується» у пам'яті так, щоб вмістити всі потрібні елементи даних. Отже, він здатен досить економічно використовувати наявну динамічну пам'ять, займаючи тільки той її об'єм, який мінімально необхідний для збереження даних.

Другою перевагою контейнерів є те, що вони здатні підтримувати виконання різноманітних операції з елементами даних[1]: додавання елементів у контейнер, видалення елементів з контейнера, вставку елементів у певне місце контейнера, підрахунок кількості елементів, що зберігаються у контейнері, сортування елементів даних тощо. Зазначені операції, як правило, реалізуються за допомогою відкритих методів. Це дозволяє легко використовувати контейнери у програмах на C++ і, у той же час, дозволяє сховати реалізацію коду контейнера всередині його методів.

На даний момент відомо багато видів контейнерів, які відрізняються способом розташування даних у динамічній пам'яті, правилами доступу до елементів контейнера, алгоритмами сортування елементів та обробки даних тощо. На практиці часто використовуються контейнери, які реалізують динамічні масиви (dynamic arrays), динамічні рядки (dynamic strings), списки (lists), стеки (stacks), множини (sets), черги (queues), хеш-таблиці (hash tables), словники (dictionaries), бінарні дерева (binary trees) тощо. Найбільш відомими серед них є динамічні масиви, динамічні рядки та списки, які будуть розглядатись далі.

[1] Звичайно, якщо програміст-розробник контейнера реалізував для нього відповідні функціональні можливості.

Динамічні масиви

Динамічним масивом називають контейнер, в якому забезпечується довільний доступ до елементів даних, що послідовно розміщуються в динамічній пам'яті. По суті, динамічні масиви є аналогами статичних масивів, що вивчались у лабораторній роботі № 4, слід лише врахувати, що дані динамічних масивів розміщуються в динамічній пам'яті.

У лабораторній роботі № 5 розглядались приклади роботи з масивами даних, що виділялись у динамічній пам'яті за допомогою функцій стандартної бібліотеки C `malloc()`, `calloc()` та `realloc()`, або за допомогою оператора C++ `new[]` (див. лістинги 5.6, 5.7). Відповідний код майже без змін може бути використаним для реалізації будь-якого динамічного масиву. Наприклад, розглянемо лістинг 8.6, в якому використовується динамічний масив комплексних чисел типу `CComplex`.

Лістинг 8.6:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <time.h>
 4: #include <math.h>
 5:
 6: ////////////////////////////////////
 7: // КЛАС, ЩО ОПИСУЄ КОМПЛЕКСНЕ ЧИСЛО.
 8: ////////////////////////////////////
 9: class CComplex
10: {
11:     double re, im;
12:
13: public:
14: // Конструктори.
15:     CComplex(): re(0.0), im(0.0) {}
16:     CComplex( double _re )
17:     {
18:         re = _re;
19:         im = 0.0;
20:     }
21:     CComplex( double _re, double _im )
22:     {
23:         re = _re;
24:         im = _im;
25:     }
26:     CComplex( const CComplex& r )
27:     {
28:         this->re = r.re;
```

```
29:           this->im = r.im;
30:       }
31:  // Деструктор.
32:       ~CComplex(){}
33:
34:  // Методи.
35:       double     Arg() const
36:       {
37:           return atan2(im, re);
38:       }
39:       double     Im() const { return im; }
40:       void       Im( double _im ){ im = _im; }
41:       double     Re() const { return re; }
42:       void       Re( double _re ){ re = _re; }
43:       CComplex&  IncBy( const CComplex& z );
44:       double     Module() const
45:       {
46:           return sqrt(re*re + im*im);
47:       }
48:       void       Print() const
49:       {
50:           printf("CComplex( %f , %f )", re, im);
51:       }
52:  };
53:
54:  CComplex& CComplex::IncBy( const CComplex& z )
55:  {
56:       re += z.re;
57:       im += z.im;
58:       return *this;
59:  }
60:
61:  // Функція для додавання об'єктів CComplex.
62:  CComplex cAdd( const CComplex& a, const CComplex&
63:  b )
64:  {
65:       return CComplex( a.Re() + b.Re(), a.Im() +
66:  b.Im() );
67:  }
68:
69:  ////////////////////////////////////////
70:  // ДИНАМІЧНИЙ МАСИВ КОМПЛЕКСНИХ ЧИСЕЛ.
71:  ////////////////////////////////////////
72:  class DArray
73:  {
```

```
 74: private:
 75: // Дані.
 76:     CComplex* p;
 77:     int       size;
 78:
 79: public:
 80: // Конструктор.
 81:     DArray()
 82:     {
 83:         p    = NULL;
 84:         size = 0;
 85:     }
 86: // Деструктор.
 87:     ~DArray()
 88:     {
 89:         if( p ) delete[] p;
 90:         p    = NULL;
 91:         size = 0;
 92:     }
 93:
 94: // Доступ до даних.
 95:     CComplex* ElementPtr( int index );
 96:
 97: // Операції з масивом.
 98:     bool Alloc( int _size );
 99:     bool ReAlloc( int _size );
100:     void Free();
101:     bool Sort();
102:     bool Fill();
103:     bool Print();
104: };
105:
106: CComplex* DArray::ElementPtr( int index )
107: {
108:     if( p && (index >=0) && (index < size) )
109:     {
110:         return p+index;
111:     }
112:     else return NULL;
113: }
114:
115: bool DArray::Alloc( int _size )
116: {
117:     if( (_size < 0) || (_size > 1000000) )
118:         return false;
```

```
119:
120:     if( p ) Free();
121:     if( _size == 0 ) return true;
122:
123:     p = new CComplex[_size];
124:     if( p )
125:     {
126:         size = _size;
127:         return true;
128:     }
129:
130:     return false;
131: }
132:
133: bool DArray::ReAlloc( int _size )
134: {
135:     if( (_size < 0) || (_size > 1000000) )
136:         return false;
137:
138:     CComplex* p2 = new CComplex[_size];
139:     if( !p2 ) return false;
140:
141:     if( p ) Free();
142:
143:     p = p2;
144:     size = _size;
145:
146:     return true;
147: }
148:
149: void DArray::Free()
150: {
151:     if( p ) delete[] p;
152:     p    = NULL;
153:     size = 0;
154: }
155:
156: bool DArray::Sort()
157: {
158:     CComplex tmp;
159:
160:     if( !p ) return false;
161:
162:     for( int i = 0; i < size; ++i )
163:     {
```

```
164:            // Отримати індекс мінімального
165:            // елемента.
166:            int iMinIndex = i;
167:            for( int j = i; j < size; ++j )
168:            {
169:                // Сортування за дійсною частиною.
170:                if( (p+j)->Re() <
171:                    (p+iMinIndex)->Re() )
172:                    iMinIndex = j;
173:            }
174:            // Якщо потрібно переставити елементи
175:            // місцями.
176:            if( iMinIndex != i )
177:            {
178:                tmp               = *(p + iMinIndex);
179:                *(p + iMinIndex) = *(p + i);
180:                *(p + i)          = tmp;
181:            }
182:        }
183:
184:        return true;
185:    }
186:
187:    bool DArray::Fill()
188:    {
189:        if( !p ) return false;
190:
191:        // Заповнити даними.
192:        CComplex* ptr = p;
193:        for( int i = 0; i < size; i++ )
194:        {
195:            ptr->Re(rand() % 100);
196:            ptr->Im(rand() % 100);
197:            ptr++;
198:        }
199:
200:        return true;
201:    }
202:
203:    bool DArray::Print()
204:    {
205:        if( !p ) return false;
206:
207:        printf("\nArray of CComplex at 0x%p:\n", p);
208:        CComplex* ptr = p;
```

```
209:      for( int i = 0; i < size; i++ )
210:      {
211:          printf("  [%1d]: {%f + i %f}\n", i,
212:              ptr->Re(), ptr->Im());
213:          ptr++;
214:      }
215:      printf("\n");
216:
217:      return true;
218:  }
219:
220:  ////////////////////
221:  // ФУНКЦІЯ main().
222:  ////////////////////
223:  int main( void )
224:  {
225:  // Ініціалізувати генератор випадкових чисел.
226:      srand( (unsigned int)time(NULL) );
227:
228:      DArray a;
229:      a.Alloc(3);
230:      a.Fill();
231:      a.Print();
232:
233:      a.ReAlloc(4);
234:      a.Fill();
235:      a.Print();
236:
237:      a.Sort();
238:      a.Print();
239:
240:      return 0;
241:  }
```

Лістинг 8.6 можна умовно розбити на три частини. У першій частині (рядки 1-67), після включення файлів заголовків, оголошується клас `CComplex` і визначаються його методи. Ця частина програми аналогічна коду, наведеному в лістингу 8.5. Друга, найбільша, частина програми займає рядки 69-218. У ній оголошується клас динамічного масиву `DArray` комплексних чисел типу `CComplex` і визначаються методи цього контейнеру. Остання частина програми включає в себе код функції `main()` (рядки 223-241).

Відразу ж звернемо увагу на те, що тіло функції `main()` містить *доволі простий* код. Весь код цієї функції займає лише 11 рядків і зміст кожного рядка є досить очевидним, виходячи з назви метода, що викликається. Але ця

простота, насправді, є ілюзорною, оскільки за кожним викликом методу виду `a.Alloc(3);` або `a.Print();` приховано доволі значний об'єм коду (див. рядки 115-131, 203-218). Тим не менш, якщо у програміста немає потреби розбиратися з внутрішньою будовою коду контейнера, він може використовувати вже готовий (кимось розроблений) контейнер і писати прості програми, схожі на тіло функції `main()`. Саме так вчиняють більшість програмістів C/C++.

Розглянемо більш детально тіло функції `main()`. Спочатку, в рядку 226 ініціалізується генератор випадкових чисел. Далі, в рядку 228, створюється об'єкт-контейнер `a` типу `DArray`. Зазначимо, що хоча сам цей контейнер – об'єкт `a` – розміщується у локальній пам'яті, дані, які він зберігає, будуть розташовуватись у динамічній пам'яті.

У рядку 229 за допомогою методу `Alloc()` виділяється динамічна пам'ять для збереження в контейнері 3-х елементів типу `CComplex`. У наступному рядку 230 відповідні елементи заповнюються випадковими даними при виклику метода `Fill()`, після чого вміст динамічного масиву виводиться на консоль за допомогою методу `Print()`.

У наступному блоці коду (рядки 233-235) розмір динамічного масиву збільшується до 4 елементів типу `CComplex`. Це відбувається завдяки виклику методу `ReAlloc()` з аргументом `4`. Далі отриманий масив знову заповнюється випадковими даними, а його вміст виводиться на консоль.

Метод `Sort()`, який викликається в рядку 237, виконує сортування елементів динамічного масиву за величиною дійсної частини комплексного числа. Далі відсортований масив виводиться на консоль за допомогою `Print()`.

Розглянемо тепер більш детально реалізацію динамічного масиву типу `DArray`. У класі `DArray` є два закритих поля (рядки 76, 77): `CComplex* p` та `int size`. Вказівник `p` є вказівником на блок динамічної пам'яті, в якому зберігаються дані контейнера, а поле `size` визначає поточну кількість доступних елементів типу `CComplex` у масиві.

Конструктор за замовчуванням `DArray()` ініціалізує вказівник `p` значенням `NULL` (тобто помічає цей вказівник як недійсний) і задає поточний розмір масиву `size` як `0`. Отже, у новоствореному об'єкті класу `DArray` блок динамічної пам'яті, в якому мають зберігатись дані, буде відсутнім.

Деструктор класу `~DArray()` виконує звільнення раніше виділеного блоку пам'яті, на якій вказує `p`, за допомогою `delete[]`. Він також задає для полів `p` та `size` значення, що відповідають відсутності виділеного блоку динамічної пам'яті.

В тілі класу (рядки 95, 98-103) оголошуються відкриті методи, які визначаються поза межами класу.

Метод `ElementPtr( int index )` повертає вказівник на елемент з індексом `index` всередині динамічного масиву, або `NULL` у випадку помилки (рядки 106-113). Спочатку, в методі перевіряється коректність вказівника `p`, а

також коректність переданого індексу `index` (він має бути у межах від `0` до `size-1`). Якщо ці дані коректні, повертається вказівник на шуканий елемент як `p+index`. Інакше – метод повертає `NULL`.

Метод `Alloc( int _size )` виділяє динамічну пам'ять під масив з `_size` елементів (рядки 115-131). Спочатку метод перевіряє коректність переданого параметру `_size`. Далі ті дані, які існували раніше, видаляються за допомогою методу `Free()`. Якщо новий розмір масиву ненульовий, виділяється новий блок даних за допомогою `new CComplex[_size]`. Якщо ця операція виділення динамічної пам'яті була успішна, вказівник на виділену блок пам'яті та новий розмір масиву зберігаються всередині контейнера.

Подібним чином працює і метод `ReAlloc( int _size )`, який змінює розмір існуючого масиву (рядки 133-147).

Метод `Free()` виконує безпечне звільнення блоку динамічної пам'яті, пов'язаному з контейнером, і задає ті значення для полів контейнера, які відповідають нульовому розміру масиву (блок динамічної пам'яті не виділений).

Найбільш складним у реалізації є метод `Sort()`, який виконує вибіркове сортування елементів масиву за величиною дійсної частини комплексного числа (рядки 156-185). Алгоритм роботи цього методу ідентичний до алгоритму роботи коду з лістингу 4.5 (лабораторна робота № 4). Єдина принципова відмінність між кодом з лістингу 8.6 та кодом з лістингу 4.5 полягає у використанні методу `Re()` для звернення до дійсної частини комплексного числа, що описується класом `CComplex`.

Метод `Fill()` заповнює динамічний масив елементами з випадковими даними. Для кожного елементу масиву дійсна та уявна частини комплексного числа ініціалізуються випадковими значеннями `rand() % 100`.

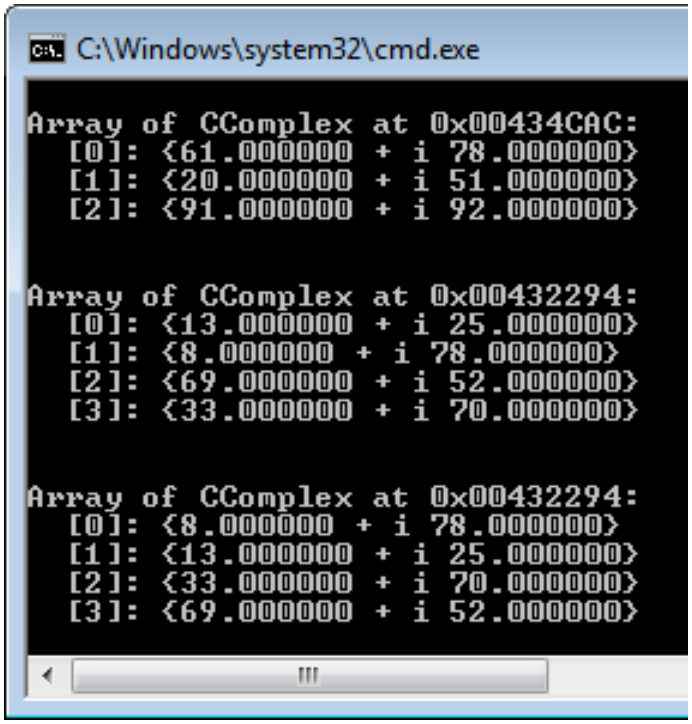


Рис. 27

Останній метод `Print()` у циклі проходить вздовж усього динамічного масиву, що адресується вказівником `p`. Для кожного елементу він виводить на консоль його вміст, використовуючи методи `Re()` та `Im()` класу `CComplex`.

Приклад текстових повідомлень, які виводяться на консоль, при роботі програми з лістингу 8.6 показано на рис. 27.

Наприкінці зазначимо,

що реалізація динамічного масиву DArray з лістингу 8.6 не є абсолютно ідеальною та єдино можливою. За необхідності відповідний код може бути помітно покращеним, наприклад, у методі ReAlloc() може бути передбачена можливість збереження вже існуючих даних, також у клас DArray можуть бути додані різні додаткові можливості (вставка/видалення елементів, сортування за вказаною умовою тощо).

Динамічні рядки

Динамічний рядок є динамічним масивом символів, здатним зберігати дані текстового рядка. Таким чином, алгоритм збереження даних у динамічному рядку принципово нічим не відрізняється від алгоритму збереження даних у динамічному масиві. Єдине що слід пам'ятати, якщо текстовий рядок задається в форматі C, то при виділенні блоку динамічної пам'яті має бути передбачено місце для збереження нульового символу '\0', яким має завершуватись рядок у форматі C. Також слід зауважити, що, на відміну від звичайних динамічних масивів (об'єктів довільного типу), динамічні рядки часто включають в себе значну кількість відкритих методів, здатних виконувати різноманітні операції з символами та рядками. Такий підхід, пов'язаний з розширенням функціональності динамічного рядка, дозволяє суттєво спростити роботу з текстовою інформацією у програмах на C++, оскільки динамічні рядки можна використовувати не тільки збереження даних, але й для додавання/вставлення/пошуку/видалення символів і символьних рядків, зміни регістру символів, інверсії, шифрування, сортування даних рядка тощо.

У лістингу 8.7 наведено приклад програми, де вводиться простий клас динамічного рядка String та ілюструється використання можливостей такого контейнера:

Лістинг 8.7:

```
 1: #define _CRT_SECURE_NO_WARNINGS
 2: #include <stdio.h>
 3: #include <string.h>
 4:
 5: class String
 6: {
 7: private:
 8: // Дані.
 9:     char* s;
10:     int   len;
11:
12: public:
13: // Конструктори.
```

```
14:      String(){ s = NULL; len = -1; }
15:      String( const char* _s )
16:      {
17:          s = NULL;
18:          len = -1;
19:
20:          if( _s )
21:          {
22:              int _len = (int)strlen(_s);
23:              s = new char[_len + 1];
24:              if( s )
25:              {
26:                  strcpy(s, _s);
27:                  len = _len;
28:              }
29:          }
30:      }
31:  // Деструктор.
32:      ~String()
33:      {
34:          Cleanup();
35:      }
36:
37:  // Методи.
38:      void Add( const char* _s );
39:
40:      void Cleanup()
41:      {
42:          if( s )
43:          {
44:              delete[] s;
45:              s = NULL;
46:          }
47:          len = -1;
48:      }
49:
50:      char* GetString() const { return s; }
51:
52:      void Print() const
53:      {
54:          if( s ) printf(s);
55:      }
56:
57:      void SetString( const char* _s );
58:  };
```

```
 59:
 60: void String::Add( const char* _s )
 61: {
 62:     if( _s )
 63:     {
 64:         int _len = (int)strlen(_s);
 65:         if( s ) _len += len;
 66:         // +1 для збереження '\0'.
 67:         char* p = new char[_len + 1];
 68:         if( p )
 69:         {
 70:             if( s )
 71:             {
 72:                 strcpy(p, s);//Скопіювати дані
 73:                 strcat(p, _s);//Додати нові дані
 74:             }
 75:             else strcpy(p, _s);//Скопіювати дані
 76:         }
 77:         if( s ) delete[] s;
 78:         s = p;
 79:         len = _len;
 80:     }
 81: }
 82:
 83: void String::SetString( const char* _s )
 84: {
 85:     if( _s )
 86:     {
 87:         int _len = (int)strlen(_s);
 88:         // +1 для збереження '\0'.
 89:         char* p = new char[_len + 1];
 90:         if( !p ) return; // Помилка!
 91:
 92:         strcpy(p, _s); // Скопіювати дані.
 93:
 94:         if( s ) delete[] s;
 95:         s = p;
 96:         len = _len;
 97:     }
 98: }
 99:
100: int main( void )
101: {
102:     String s1("\nHello");
103:     String s2;
```

```
104:      s2.SetString(", ");
105:      s2.Add("young programmer!\n\n");
106:      s1.Add(s2.GetString());
107:      s1.Print();
108:
109:      s1.Cleanup();
110:      s2.Cleanup();
111:
112:      return 0;
113:  }
```

Директива препроцесора `#define _CRT_SECURE_NO_WARNINGS` (рядок 1) забороняє компілятору виведення повідомлень про наявність більш безпечних версій стандартних функцій `strcpy()` та `strcat()`, які використовуються у програмі. Щоб мати можливість звертатись до функцій `strcpy()` та `strcat()`, в програму включається файл заголовку `<string.h>`.

Клас-контейнер динамічного рядка `String` оголошується в рядках 5-58. Цей клас має два закритих поля. Перше поле `char* s` є вказівником на блок динамічної пам'яті, в якому зберігаються дані текстового рядка. Якщо динамічна пам'ять не була виділена, вказівник `s` має бути нульовим (недійсним). Друге поле `int len` містить інформацію про довжину рядка без урахування нульового символу `'\0'`. Якщо динамічна пам'ять для рядка не виділена, в поле `len` записується значення `-1` (див. конструктор за замовчуванням у рядку 14). Значення `0` для `len` означає, що рядок є пустим – він не містить жодного символу, окрім `'\0'`. Додатні значення `len` визначають кількість змістовних символів у динамічному рядку.

На відміну від динамічних масивів, для динамічних рядків прийнято визначати кілька конструкторів. У лістингу 8.7 наведено код лише для конструктора за замовчуванням (рядок 14) та конструктора, що ініціалізує об'єкт класу текстовим рядком, що передається у вигляді параметра `const char* _s` (рядки 15-30).

Всередині конструктора `String( const char* _s )` для полів `s` та `len` спочатку задаються значення, що відповідають випадку відсутності даних (рядки 17, 18). Потім, якщо параметр конструктора `_s` є ненульовим, для збереження даних виділяється блок динамічної пам'яті потрібного розміру (рядки 20-23). І, якщо виділення пам'яті було успішним, у цей блок пам'яті переносяться дані з текстового рядка `_s` (рядки 24-28).

Деструктор `~String()` просто викликає метод `Cleanup()`, який виконує роботу по звільненню ресурсів контейнера. Цей метод, реалізований у рядках 40-48, звільняє раніше виділений блок пам'яті за допомогою оператора `delete[]` і записує в полія `s` та `len` значення, характерні для пустого контейнера.

Окрім метода `Cleanup()`, у класі `String` є ще кілька методів. Метод `GetString()` повертає вказівник на текстовий рядок, що зберігається в контейнері (рядок 50). Метод `Print()`, визначений у рядках 52-55, виводить вміст текстового рядка, що зберігається в контейнері, на консоль. Інші два методи, `Add( const char* _s )` та `SetString( const char* _s )`, оголошуються в рядках 38 та 57 і визначаються ззовні від оголошення класу.

Метод `Add( const char* _s )` додає до текстового рядка, що зберігається в контейнері, інший текстовий рядок, на який вказує параметр-вказівник `_s`. Цей метод спочатку розраховує розмір блоку динамічної пам'яті, потрібний для збереження існуючих даних та даних того текстового рядка, який додається (рядки 64-65). Потім виділяється блок динамічної пам'яті відповідного розміру (рядок 67). Якщо виділення пам'яті пройшло успішно, в цей блок пам'яті переносяться вже існуючі дані контейнера (за допомогою `strcpy()`), а потім – нові дані з текстового рядка `_s` (за допомогою `strcat()`, див. рядки 68-74). Останнім кроком код методу `Add()` звільняє той блок динамічної пам'яті `s`, який раніше використовувався контейнером. Також він встановлює поля контейнера так, щоб вони відповідали новому, тільки-но виділеному блоку пам'яті, заповненому потрібними даними (рядки 77-79).

Робота методу `SetString( const char* _s )` відбувається за схожим сценарієм (рядки 83-98). Спочатку виділяється новий блок динамічної пам'яті з розміром, достатнім для збереження переданого методу текстового рядка (рядки 87-90). Потім у цей блок пам'яті копіюються дані з переданого текстового рядка (рядок 92) і відповідні дані зберігаються в полях класу (рядки 94-96).

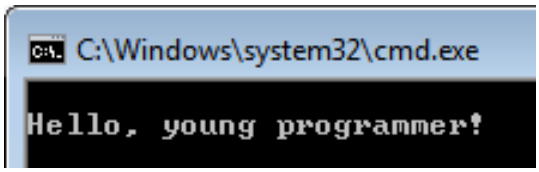


Рис. 28

Результат роботи програми з лістингу 8.7 показано на рис. 28.

Підкреслимо, що наведена реалізація динамічного рядка не є оптимальною та єдино можливою. За бажанням відповідний код можна суттєво вдосконалити.

Списки

Список – це контейнер, в якому елементи (або ***вузли***) утворюють впорядковану послідовність, але їх порядок визначається не індексом елементу (як у масиві), а вказівниками.

Для реалізації списків, зазвичай, використовують два класи (структури). Перший клас – це, власне, сам клас-контейнер списку, в якому зберігається вказівник на певний елемент списку або кілька вказівників на елементи списку. Другий клас (структура) – це структурований тип, що описує елемент-вузол списку. Кожен елемент списку повинен містити не тільки корисні дані (які зберігаються в цьому елементі), але й вказівник на інший елемент списку

(або вказівники на інші елементи). Завдяки цьому для кожного типу даних, що мають зберігатись у списку, як правило, доводиться створювати свій тип вузла списку, а також свій тип контейнера-списку, якій уміє працювати з вузлами зазначеного типу.

Списки можуть бути *однозв'язними* та *двозв'язними*.

Однозв'язний або ***простий список*** – це список, у якому елементи зв'язані між собою за допомогою одного вказівника. За традицією цей вказівник вводять як вказівник на *наступний* елемент списку. Наприклад, елемент однозв'язного списку може бути реалізований як наступна структура:

```
struct ListItem
{
    // Дані, які має зберігати елемент списку...

    // Вказівник на наступний елемент.
    ListItem* pNext;
};
```

Використовуючи вказівник на перший елемент однозв'язного списку – т. зв. ***голову списку*** (цей вказівник зберігається всередині контейнера-списку), а також вказівники, задані всередині елементів списку (вказівники на наступні елементи), можна послідовно проглядати/перебирати всі елементи списку, від першого до останнього (рис. 29). Ознакою останнього елементу списку може бути те, що його вказівник на наступний елемент позначено як недійсний (NULL).

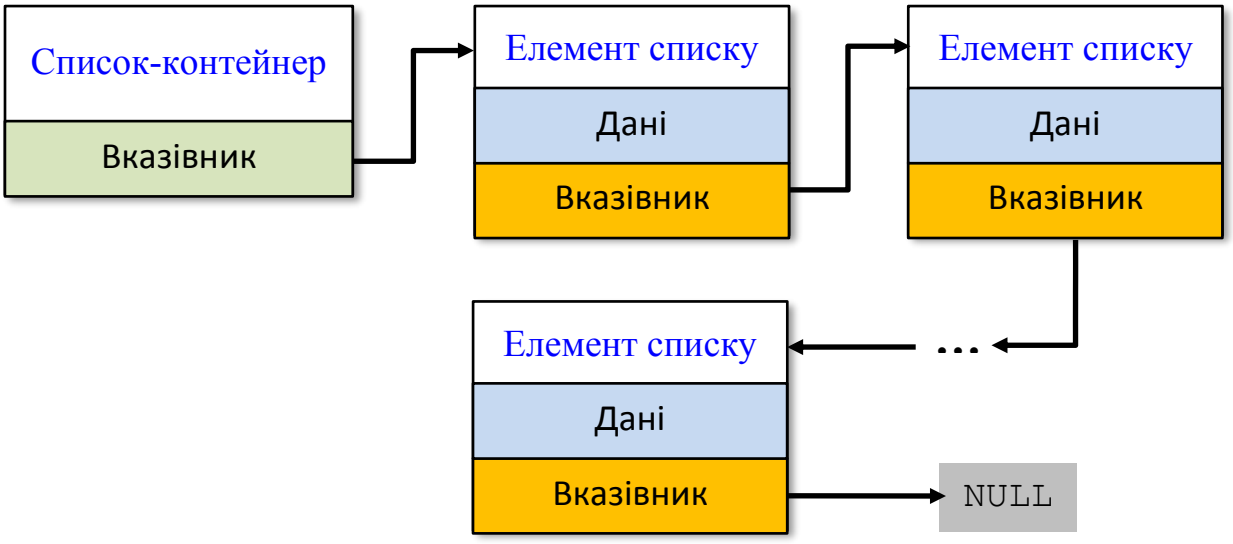


Рис. 29

Наступний фрагмент коду демонструє яким чином можна «проходити» вздовж однозв'язного списку, по черзі перебираючи всі його вузли (типу

ListItem):

```
// pHead -- вказівник на голову списку.
ListItem* p = pHead;
while( p ) // Поки p != NULL.
{
    // Виконуємо певні дії з елементом списку...
    p = p->pNext; // Переходимо до наступного елементу.
}
```

Окрім однозв'язних списків часто застосовуються ***складні*** або ***двозв'язні списки***, в яких кожен вузол пов'язується через вказівники з двома сусідніми вузлами – попереднім та наступним. За рахунок такого «подвійного» зв'язку між елементами, у двозв'язному списку можна «переміщуватись» вздовж списку як в прямому напрямку (від першого елементу до останнього, використовуючи вказівник на наступний елемент), так і в зворотному напрямку (від останнього елементу до першого, використовуючи вказівник на попередній елемент). Відповідно, елемент двозв'язного списку може бути реалізований як наступна структура:

```
struct DoubleListItem
{
    // Дані, які має зберігати елемент списку...

    // Вказівник на попередній елемент.
    DoubleListItem* pPrev;
    // Вказівник на наступний елемент.
    DoubleListItem* pNext;
};
```

Для того щоб спростити використання двозв'язних списків до класу-контейнеру списку, як правило, включають два вказівники. Один з них вказує на ***голову списку*** – тобто на перший елемент у списку, а іншій – на останній елемент у списку – на його ***хвіст***. Якщо список пустий (не містить жодного елементу), обидва ці вказівники, зазвичай, ініціалізуються значенням NULL (тобто помічаються як недійсні).

У лістингу 8.8 наведено приклад створення і використання однозв'язного списку цілих чисел:

Лістинг 8.8:

```
1: #include <stdio.h>
2:
3: // Елемент однозв'язного списку цілих чисел.
4: struct SListItem   // S = Single
```

```
 5: {
 6:     int n;
 7:     SListItem* pNext;
 8:
 9:     SListItem()
10:     {
11:         n = 0;
12:         pNext = NULL;
13:     }
14:     SListItem( int _n )
15:     {
16:         n = _n;
17:         pNext = NULL;
18:     }
19: };
20: // Контейнер однозв'язного списку цілих чисел.
21: class SList
22: {
23: private:
24:     // Вказівник на голову списку.
25:     SListItem* pHead;
26:
27: public:
28:     SList()
29:     {
30:         pHead = NULL;
31:     }
32:     ~SList()
33:     {
34:         Cleanup();
35:     }
36:
37:     void Cleanup();
38:
39:     void AddHead( int _n );
40:     void RemoveHead();
41:
42:     void Print();
43: };
44:
45: void SList::Cleanup()
46: {
47:     SListItem* p = pHead;
48:     SListItem* p2;
49:     while( p )
```

```
50:     {
51:         p2 = p->pNext;
52:         delete p;
53:         p = p2;
54:     }
55:     pHead = NULL;
56: }
57:
58: void SList::AddHead( int _n )
59: {
60:     SListItem* p = new SListItem(_n);
61:     if( p )
62:     {
63:         p->pNext = pHead;
64:         pHead = p;
65:     }
66: }
67:
68: void SList::RemoveHead()
69: {
70:     if( pHead )
71:     {
72:         SListItem* p = pHead->pNext;
73:         delete pHead;
74:         pHead = p;
75:     }
76: }
77:
78: void SList::Print()
79: {
80:     SListItem* p = pHead;
81:     int i = 0;
82:     while( p )
83:     {
84:         printf("Element {%d} is %d\n", i, p->n);
85:         p = p->pNext;
86:         i++;
87:     }
88:     printf("\n");
89: }
90:
91: int main( void )
92: {
93:     SList list;
94:     list.AddHead(10);
```

```
 95:       list.AddHead(20);
 96:       list.AddHead(30);
 97:       list.Print();
 98:
 99:       list.RemoveHead();
100:       list.Print();
101:
102:       return 0;
      }
```

У лістингу 8.8 вводяться два нових типи: `SListItem` та `SList`.

Структура `SListItem` (рядки 4-19) описує вузол списку[1]. Вона включає в себе поле з корисними даними (`int n`) та вказівник `SListItem* pNext` на наступний вузол.

Для цієї структури реалізуються конструктор за замовчуванням `SListItem()` та спеціалізований конструктор `SListItem( int _n )`, який ініціалізує поле даних вузла вказаним значенням `_n`. Код обох конструкторів встановлює вказівник на наступний елемент списку `pNext` як `NULL`.

Для структури `SListItem` не записано деструктор, тому компілятор реалізує його автоматично – як пустий деструктор.

Основним компонентом програми є клас `SList`, який реалізує контейнер однозв'язного списку. Клас має єдине закрите поле – вказівник на голову списку `pHead`, а також конструктор за замовчуванням `SList()`, деструктор `~SList()` і кілька відкритих методів.

Конструктор `SList()` ініціалізує поле `pHead` значенням `NULL`, помічаючи вказівник `pHead` як недійсний (рядки 28-31). Деструктор класу викликає метод `Cleanup()`, який виконує видалення всіх елементів зі списку (рядки 32-35).

Метод `Cleanup()`, визначений у рядках 45-56, проходить за допомогою циклу `while` вздовж усього списку і по черзі видаляє елементи зі списку, починаючи з першого (на який вказує `pHead`) і закінчуючи останнім. Для кожного існуючого вузла, на який вказує вказівник `p`, виконується тіло циклу:

```
p2 = p->pNext;// Отримати вказівник на наступний вузол.
delete p;     // Видалити поточний вузол.
p = p2;       // Перейти до наступного вузла.
```

Після цього код `Cleanup()` помічає внутрішній вказівник `pHead` як

[1] Тип вузла списку `SListItem` в наведеній програмі відіграє достатньо другорядну роль, тому, для спрощення, він реалізується як структура, а не як клас. До того ж оголошення структури `SListItem` демонструє, що у мові C++ структури можуть мати елементи-функції, так само як і класи.

недійсний.

Метод `AddHead( int _n )` додає на початок списку новий вузол-елемент і ініціалізує його дані вказаною величиною `_n`. Для цього виділяється пам'ять під новий елемент типу `SListItem`, який автоматично ініціалізується за допомогою конструктора `SListItem( int _n )` (рядок 60). Якщо виділення пам'яті відбулось успішно, відповідний елемент підключається до голови списку. Підключення відбувається за допомогою двох рядків коду (рядки 63, 64). Спочатку поле `pNext` всередині новоствореного елементу задається як вказівник на поточну голову списку `pHead`. Потім вказівник `pHead` встановлюється як вказівник на новостворений елемент.

Метод `RemoveHead()` виконує зворотну задачу – цей метод видаляє перший елемент у списку. Алгоритм роботи цього методу наступний. Спочатку як `SListItem* p = pHead->pNext;` визначається адреса елементу, наступного за першим. Потім видаляється перший елемент зі списку (`delete pHead;`) і перевизначається вказівник на перший елемент (`pHead = p;`). Після такої операції вказівник `pHead` вже вказує на той елемент, який раніше був наступним після першого, і який, після видалення першого елементу, вже сам став першим елементом у списку.

Метод `Print()` проходить вздовж усього списку і виводить на консоль вміст даних кожного з його вузлів (поля `n` структури `SListItem`). Принципова відмінність цього коду від аналогічного коду для динамічного масиву полягає в тому, що адреса вузла, який іде наступним після поточного, визначається в тілі циклу як `p = p->pNext`.

Використання запропонованого списку ілюструється в тілі функції `main()` – див. рядки 93-100. Спочатку створюється об'єкт `list` типу `SList`, який являє собою пустий список. Потім за допомогою `AddHead()` до списку послідовно додаються елементи, що містять числа `10`, `20` та `30`. Вміст списку виводиться на консоль за допомогою виклику методу `list.Print();`.

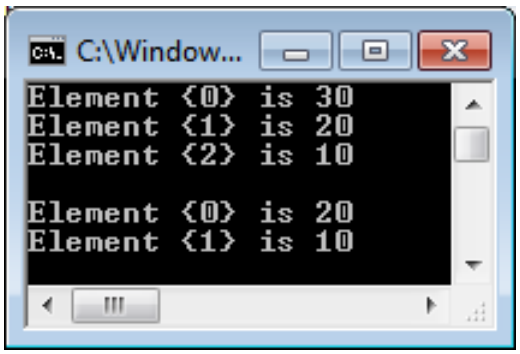


Рис. 30

У наступному блоці коду (рядки 99, 100) перший елемент списку видаляється і вміст списку знову виводиться на консоль за допомогою `Print()`.

Програма лістингу 8.8 виводить на консоль інформацію, показану на рис. 30.

Зауважимо, що в тілі функції `main()` явним чином не викликається метод `Cleanup()` для звільнення динамічної пам'яті, раніше виділеної під елементи списку. Відповідна операція виконується автоматично під час виклику деструктора класу, коли знищується локальний об'єкт `list`.

Наприкінці зазначимо, що перевагою списків порівняно з динамічними

масивами є можливість швидко і з малими затратами процесорного часу додавати та видаляти елементи в довільному місці списку. Також слід врахувати, що при використанні списків динамічна пам'ять виділяється окремо під кожний елемент списку – блоками достатньо малого розміру, а не одним великим блоком як у динамічному масиві. Відповідно, програма, що використовує списки більш надійно працює в умовах нестачі динамічної пам'яті, ніж програма, яка використовує для збереження даних динамічні масиви. Натомість, суттєвою перевагою динамічних масивів є можливість швидкого поіндексного доступу до його елементів, у той час, як у списках для знаходження потрібного елементу слід проходити уздовж всього списку, шукаючи потрібний елемент.

Завдання

Варіант 1

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* чисел типу `int`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран результат поелементного додавання даних масивів M1 та M2 (тобто `M1[0]+M2[0]`, `M1[1]+M2[1]`, …).
6) Розробити клас-контейнер динамічного рядка.
7) Задати вміст екземпляра динамічного рядка S як символьне представлення набору чисел, що зберігаються у масивах M1 та M2, записаних через пробіл `' '`.
8) Вивести вміст динамічного рядка S на консоль.
9) Розробити клас-контейнер однозв'язного списку.
10) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як суму елементів масивів M1 та M2 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
11) Вивести на екран вміст заповненого списку L.
12) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 2

1) Розробити клас-контейнер динамічного масиву та клас-контейнер динамічного рядка, що може зберігати символьний рядок у форматі C.

2) Створити:
а) екземпляр динамічного рядка, який містить всі дні тижня, записані послідовно «Понеділок_Вівторок_Середа_….._Неділя» (для назв днів тижня можна, за бажанням, використовувати англійську мову);
б) екземпляр динамічного масиву, в якому зберігається послідовність цілих чисел від 0 до 23.
3) Реалізувати функцію, яка буде виводити на консоль таблицю доступних станів певної системи (стани «enable» та «disable») за весь тиждень, використовуючи звернення до елементів динамічного масиву та динамічного рядка. Вважати, що система починає працювати з понеділка з 0:00 зі стану disable і стан системи змінюється кожні 5 годин. При виведенні тексту назви днів тижня зчитуються з динамічного рядка, а час зчитується з динамічного масиву. Приклад роботи програми:
```
Графік роботи:
Понеділок enable [0.00:5.00], disable [5.00:10.00],
… enable [20.00:1.00]
Вівторок disable [1.00:6.00], enable [6.00:10.00], …
enable [21.00:3.00] …
```
4) Реалізувати функцію, яка за введеними з клавіатури командами буде вносити зміни у графік, що потім відображається на консолі. Наприклад, змінює початковий момент відліку та стан системи:
```
Введіть початок відліку та початковий стан: 2, disable
Понеділок disable [2.00:7.00] enable [7.00:12.00] …
enable [22.00:3.00] …
```
5) Розробити клас-контейнер однозв’язного списку, що може зберігати цілі числа.
6) Реалізувати пп. 2-4, використовуючи не динамічний масив цілих чисел, а однозв’язний список цілих чисел від 0 до 23.
7) Провести звільнення всіх блоків виділеної динамічної пам’яті перед завершенням функції `main()` (до оператора `return`).

Варіант 3

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить N чисел типу `int`. Число $N = 5 – 10$ вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран результат поелементного віднімання даних масивів M1 та M2 (тобто `M1[0]-M2[0]`, `M1[1]-M2[1]`, …).
6) Розробити клас-контейнер динамічного рядка.

7) Задати вміст екземпляра динамічного рядка S як символьне представлення набору чисел, що зберігаються у масивах M1 та M2, записаних через пробіл `' '`.
8) Вивести вміст динамічного рядка S на консоль.
9) Розробити клас-контейнер однозв'язного списку.
10) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як різницю елементів масивів M1 та M2 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
11) Вивести на екран вміст заповненого списку L.
12) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 4

1) Розробити клас-контейнер динамічного масиву та клас-контейнер динамічного рядка, що може зберігати символьний рядок у форматі C.
2) Створити:
   а) екземпляр динамічного рядка, який містить всі дні тижня, записані послідовно «Понеділок_Вівторок_Середа_….._Неділя» (для назв днів тижня можна, за бажанням, використовувати англійську мову);
   б) екземпляр динамічного масиву, в якому зберігається послідовність цілих чисел від 0 до 23.
3) Реалізувати функцію, яка буде виводити на консоль таблицю доступних станів певної системи (стани «enable» та «disable») за весь тиждень, використовуючи звернення до елементів динамічного масиву та динамічного рядка. Вважати, що система починає працювати з понеділка з 2:30 зі стану enable і стан системи змінюється кожні 4 години. При виведенні тексту назви днів тижня зчитуються з динамічного рядка, а час зчитується з динамічного масиву. Приклад роботи програми:
```
Графік роботи:
Понеділок enable [2.30:6.30], disable [6.30:10.30],
… enable [22.30:2.30]
Вівторок disable [2.30:6.30], enable [6.30:10.30], …
enable [22.30:2.30] …
```
4) Реалізувати функцію, яка, за введеними з клавіатури командами, буде вносити зміни у графік, що потім відображається на консолі. Наприклад, змінює початковий момент відліку та стан системи:
```
Введіть початок відліку та початковий стан: 3, disable
Понеділок disable [3.00:7.00] enable [7.00:11.00] …
enable [23.00:3.00] …
```
5) Розробити клас-контейнер однозв'язного списку, що може зберігати цілі числа.
6) Реалізувати пп. 2-4, використовуючи не динамічний масив цілих чисел, а однозв'язний список цілих чисел від 0 до 23.

7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 5

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* чисел типу `float`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран результат поелементного добутку даних масивів M1 та M2 (тобто `M1[0]*M2[0]`, `M1[1]*M2[1]`, …).
6) Розробити клас-контейнер динамічного рядка.
7) Задати вміст екземпляра динамічного рядка S як символьне представлення набору чисел, що зберігаються у масивах M1 та M2, записаних через пробіл `' '`.
8) Вивести вміст динамічного рядка S на консоль.
9) Розробити клас-контейнер однозв'язного списку.
10) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як добуток елементів масивів M1 та M2 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
11) Вивести на екран вміст заповненого списку L.
12) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 6

1) Розробити клас-контейнер динамічного масиву та клас-контейнер динамічного рядка, що може зберігати символьний рядок в форматі C.
2) Створити:
   а) екземпляр динамічного рядка, який містить всі дні тижня, записані послідовно «Понеділок*Вівторок*Середа*.....*Неділя» (для назв днів тижня можна, за бажанням, використовувати англійську мову);
   б) екземпляр динамічного масиву, в якому зберігається послідовність цілих чисел від 0 до 23.
3) Реалізувати функцію, яка буде виводити на консоль таблицю доступних станів певної системи (стани «enable» та «disable») за весь тиждень, використовуючи звернення до елементів динамічного масиву та динамічного рядка. Вважати, що система починає працювати з понеділка з 1:15 зі стану disable і стан системи змінюється кожні 3 години. При виведенні тексту назви днів тижня зчитуються з динамічного рядка, а час зчитується

з динамічного масиву. Приклад роботи програми:

```
Графік роботи:
Понеділок disable [1.15:4.15], enable [4.15:7.15], …
disable [22.15:1.15]
Вівторок disable [1.15:4.15], enable [4.15:7.15], …
disable [22.15:1.15] …
```

4) Реалізувати функцію, яка за введеними з клавіатури командами буде вносити зміни у графік, що потім відображається на консолі. Наприклад, змінює початковий момент відліку та стан системи:

```
Введіть початок відліку та початковий стан: 2, enable
Понеділок enable [2.15:5.15] disable [5.15:8.15] …
enable [23.15:2.15] …
```

5) Розробити клас-контейнер однозв'язного списку, що може зберігати рядки у форматі C.
6) Реалізувати пп. 2-4, використовуючи не динамічний рядок, а однозв'язний список рядків у форматі C, кожен з яких зберігає назву дня тижня.
7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 7

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* чисел типу `double`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран результат поелементного ділення даних масивів M1 та M2 (тобто `M1[0]/M2[0]`, `M1[1]/M2[1]`, …).
6) Розробити клас-контейнер динамічного рядка.
7) Задати вміст екземпляра динамічного рядка S як символьне представлення набору чисел, що зберігаються у масивах M1 та M2, записаних через пробіл `' '`.
8) Вивести вміст динамічного рядка S на консоль.
9) Розробити клас-контейнер однозв'язного списку.
10) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як відношення елементів масивів M1 та M2 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
11) Вивести на екран вміст заповненого списку L.
12) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 8

1) Створити структуру `Data` з чотирма полями:
   а) перше поле структури містить назву дня тижня у вигляді динамічного рядка у форматі C;
   б) друге поле містить стан системи off або on, який не змінюється протягом 4-х годинного інтервалу часу;
   в) третє поле має містити час, коли система переходить у стан, визначений у другому полі. Наприклад, Понеділок, off, 4, Вівторок, on, 20 тощо;
   г) четверте поле має містити вказівник на структуру цього ж типу.
2) Розробити клас-контейнер динамічного масиву.
3) Створити динамічний масив `a` структур `Data`, в яких останнє поле вказує на наступний елемент у масиві або містить `NULL` для останнього елементу масиву. Значення інших полів заповнювати довільними, але логічно узгодженими даними.
4) Використовуючи масив `a` симулювати роботу однозв'язного списку, в якому зберігається інформація про стан деякої системи протягом одного календарного тижня.
5) Реалізувати функцію, яка буде виводити на консоль таблицю доступних станів системи за тиждень, наприклад:

```
Понеділок
0-4  on
4-8  off
8-12 on
…
Вівторок
0-4  off
4-8  off
8-12 on
…
```

6) Реалізувати роботу із заданою інформацією про стан системи за допомогою класу-контейнера однозв'язного списку. Виконати завдання п. 5 використовуючи однозв'язний список.
7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 9

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що

вводяться з клавіатури.

4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран в рядок по черзі символи з масивів M1 та M2, послідовно чергуючи їх між собою (тобто `M1[0]`, `M2[0]`, `M1[1]`, `M2[1]`, …).
6) Розробити клас-контейнер динамічного рядка.
7) Задати вміст екземпляра динамічного рядка S як рядка, що виникає при виконанні п. 5.
8) Вивести вміст динамічного рядка S на консоль.
9) Розробити клас-контейнер однозв'язного списку.
10) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як комбінацію двох символів з масивів M1 та M2 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
11) Вивести на екран вміст заповненого списку L.
12) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 10

1) Створити структуру `Data` з чотирма полями:
   а) перше поле структури містить назву дня тижня у вигляді динамічного рядка у форматі C;
   б) друге поле містить стан системи off або on, який не змінюється протягом 6-и годинного інтервалу часу;
   в) третє поле має містити час, коли система переходить у стан, визначений у другому полі. Наприклад, Понеділок, off, 4, Вівторок, on, 20 тощо;
   г) четверте поле має містити вказівник на структуру цього ж типу.
2) Розробити клас-контейнер динамічного масиву.
3) Створити динамічний масив `a` структур `Data`, в яких останнє поле вказує на наступний елемент у масиві або містить `NULL` для останнього елементу масиву. Значення інших полів заповнювати довільними, але логічно узгодженими даними.
4) Використовуючи масив `a` симулювати роботу однозв'язного списку, в якому зберігається інформація про стан деякої системи протягом одного календарного тижня.
8) Реалізувати функцію, яка буде виводити на консоль таблицю доступних станів системи за тиждень, наприклад:

```
Понеділок
0-6   on
6-12  off
12-18 on
…
Вівторок
0-6   off
6-12  off
```

```
12-18 on
```
…

5) Реалізувати роботу із заданою інформацією про стан системи за допомогою класу-контейнера однозв'язного списку. Виконати завдання п. 5 використовуючи однозв'язний список.
6) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 11

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран в рядок по черзі символи з масивів M2 та M1, послідовно чергуючи їх між собою (тобто `M2[0]`, `M1[0]`, `M2[1]`, `M1[1]`, …).
6) Реалізувати функцію, яка визначає, скільки разів деякий заданий символ входить до масивів M1 або M2. Продемонструвати результат її роботи для масивів M1 та M2.
7) Розробити клас-контейнер динамічного рядка.
8) Задати вміст екземпляра динамічного рядка S як рядка, що виникає при виконанні п. 5.
9) Вивести вміст динамічного рядка S на консоль.
10) Розробити клас-контейнер однозв'язного списку.
11) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як комбінацію двох символів з масивів M2 та M1 з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
12) Вивести на екран вміст заповненого списку L.
13) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 12

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.

4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран в рядок по черзі символи з масивів M1 та M2, послідовно чергуючи їх між собою (тобто `M1[0]`, `M2[0]`, `M1[1]`, `M2[1]`, …). Символи-букви з масиву M2 повинні виводитись як великі.
6) Реалізувати функцію, яка визначає, скільки разів деяка задана буква входить до масивів M1 або M2, як у вигляді малих, так і великих букв. Продемонструвати результат її роботи для масивів M1 та M2.
7) Розробити клас-контейнер динамічного рядка.
8) Задати вміст екземпляра динамічного рядка S як рядка, що виникає при виконанні п. 5.
9) Вивести вміст динамічного рядка S на консоль.
10) Розробити клас-контейнер однозв'язного списку.
11) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як комбінацію двох символів з масивів M1 та M2 (всі букви мають бути великими) з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.
12) Вивести на екран вміст заповненого списку L.
13) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 13

1) Розробити клас-контейнер динамічного масиву.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран в рядок по черзі символи з масивів M1 та M2, послідовно чергуючи їх між собою (тобто `M1[0]`, `M2[0]`, `M1[1]`, `M2[1]`, …).
6) Реалізувати функцію, яка визначає позицію першого входження деякого заданого символу в масиві M1 або M2. Продемонструвати результат її роботи для масивів M1 та M2.
7) Розробити клас-контейнер динамічного рядка.
8) Задати вміст екземпляра динамічного рядка S як рядка, що виникає при виконанні п. 5.
9) Вивести вміст динамічного рядка S на консоль.
10) Розробити клас-контейнер однозв'язного списку.
11) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як комбінацію двох символів з масивів M1 та M2 (всі букви мають бути великими) з однаковими індексами *n*–1; *n* приймає значення від 1 до *N*.

12) Вивести на екран вміст заповненого списку L.
13) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 14

1) Створити структурований тип даних з назвою `ListItem`, що містить чотири поля: поле `value` є статичним масивом символів `char` з кількістю елементів, що дорівнює 10; поле `length` має тип `int`; поле `repeat` має тип `bool`, а останнє поле `pointer`, має тип `ListItem*`.
2) Розробити клас-контейнер однозв'язного списку елементів `ListItem`.
3) Створити однозв'язний список з 5 елементів типу `ListItem`. Поле кожного елементу `value` заповнити *N* символами (*N* = 5 – 9, вміст поля має бути різним для кожного елементу списку). В поле `length` занести реальну довжину текстового рядка, записаного в даному елементі.
4) Реалізувати функцію, яка визначає, чи є в полі `value` відповідного елементу списку однакові символи. Занести відповідну інформацію у поле `repeat` кожного елемента.
5) Вивести на екран вміст однозв'язного списку (всі символи поля `value` в один рядок, довжину текстового рядка, та інформацію про те, чи є в ньому однакові елементи).
6) Виконати пп. 1 – 5 використовуючи збереження елементів даних у динамічному масиві, причому до кожного елементу має входити лише поле `repeat` і динамічний рядок (який містить символьний рядок у форматі C та його довжину – аналог поля `length`).
7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 15

1) Створити структурований тип даних з назвою `ListItem`, що містить чотири поля: поля `value` та `ivalue` є статичними масивами символів `char` з кількістю елементів, що дорівнює 10; поле `length` має тип `int`; останнє поле `pointer`, має тип `ListItem*`.
2) Розробити клас-контейнер однозв'язного списку елементів `ListItem`.
3) Створити однозв'язний список з 6 елементів типу `ListItem`. Поле кожного елементу `value` заповнити *N* символами (*N* = 5 – 9, вміст поля має бути різним для кожного елементу списку). В поле `length` занести реальну довжину текстового рядка, записаного в даному елементі.
4) Реалізувати функцію, яка переписує символьний рядок задом наперед (наприклад, перетворює `abcd` у `dcba`). Записати в поле `ivalue` кожного елементу результат такого «обернення» поля `value`.
5) Вивести на екран вміст однозв'язного списку (всі символи полів `value`

та `ivalue` в два окремих рядка та довжину текстового рядка).

6) Виконати пп. 1 – 5 використовуючи для збереження елементів даних динамічний масив, причому до кожного елементу має входити лише два динамічних рядка, кожен з яких містить символьний рядок у форматі C та його довжину (аналог поля `length`).
7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 16

1) Створити структурований тип даних з назвою `ListItem`, що містить чотири поля: поля `value` та `newvalue` є статичними масивами символів `char` з кількістю елементів, що дорівнює 10; поле `length` має тип `int`; останнє поле `pointer`, має тип `ListItem*`.
2) Розробити клас-контейнер однозв'язного списку елементів `ListItem`.
3) Створити однозв'язний список з 5 елементів типу `ListItem`. Поле кожного елементу `value` заповнити *N* символами (*N* = 5 – 9, вміст поля має бути різним для кожного елементу списку, але в кожному рядку має бути мінімум одна буква a). В поле `length` занести реальну довжину текстового рядка, записаного в даному елементі.
4) Реалізувати функцію, яка замінює у символьному рядку один певний символ на інший (початковий символ та символ, на який заміняється початковий, вказуються в аргументі функції). Використовуючи цю функцію записати в поле `newvalue` кожного елементу символьний рядок з `value`, в якому всі символи a замінено на b.
5) Вивести на екран вміст однозв'язного списку (всі символи полів `value` та `newvalue` в два окремих рядка та довжину текстового рядка).
6) Виконати пп. 1 – 5 використовуючи для збереження елементів даних динамічний масив, причому до кожного елементу має входити лише два динамічних рядка, кожен з яких містить символьний рядок у форматі C та його довжину (аналог поля `length`).
7) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 17

1) Розробити клас-контейнер динамічного масиву, який міг би використовуватись як динамічний рядок.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N*+1 елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури. Останній елемент у масиві повинен містити нульовий символ `'\0'`.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти:

автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.

4) Вивести на екран вміст масивів M1 та M2.
5) Вивести на екран в рядок по черзі символи з масивів M1 (в прямому порядку) та M2 (в зворотному порядку), послідовно чергуючи їх між собою (тобто `M1[0]`, `M2[N]`, `M1[1]`, `M2[N-1]`, …).
6) Зберегти результат виконання п. 5 в динамічному рядку. Вивести вміст динамічного рядка на консоль.
7) Розробити клас-контейнер однозв'язного списку.
8) Створити екземпляр L однозв'язного списку. Задати вміст кожного елементу списку з номером *n* як комбінацію двох символів з масивів M1 та M2 з індексами *n* та *N*–*n*; *n* приймає значення від 0 до *N*.
9) Вивести на екран вміст заповненого списку L.
10) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 18

1) Розробити клас-контейнер динамічного масиву, який міг би використовуватись як динамічний рядок.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N*+1 елементів типу `char`. Число *N* = 5 – 10 вводиться користувачем з клавіатури. Останній елемент у масиві повинен містити нульовий символ `'\0'`.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Створити динамічний масив M3, який містить дані масиву M1, після яких йдуть дані масиву M2.
6) Вивести на екран вміст масиву M3.
7) Реалізувати метод, який розширює розмір масиву на 1 елемент та вставляє введений з клавіатури символ у довільне місце початкового масиву. Застосувати даний метод до масиву M1 три рази та до масиву M2 два рази.
8) Повторити виконання пп. 4-6.
9) Розробити клас-контейнер однозв'язного списку.
10) Зберегти результат виконання п. 5, 8 у динамічному списку. Вивести його вміст на консоль.
11) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 19

1) Розробити клас-контейнер динамічного масиву.

2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 9 вводиться користувачем з клавіатури. Останній елемент у масиві повинен містити нульовий символ `'\0'`.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Створити динамічний масив M3, який містить дані масиву M1, після яких йдуть дані масиву M2, записані в зворотному порядку (першим – останній символ, потім передостанній тощо).
6) Вивести на екран вміст масиву M3.
7) Реалізувати метод, який видаляє символ у довільному місці початкового масиву, не змінюючи при цьому розмір самого масиву. Застосувати даний метод до масиву M1 три рази та до масиву M2 два рази.
8) Повторити виконання пп. 4-6.
9) Розробити клас-контейнер однозв'язного списку.
10) Зберегти результат виконання п. 5, 8 в динамічному списку. Вивести його вміст на консоль.
11) Провести звільнення всіх блоків виділеної динамічної пам'яті перед завершенням функції `main()` (до оператора `return`).

Варіант 20

1) Розробити класи-контейнери динамічного масиву, динамічного рядка та однозв'язного списку.
2) Задати два екземпляри динамічних масивів з іменами M1 та M2, кожен з яких містить *N* елементів типу `char`. Число *N* = 5 – 9 вводиться користувачем з клавіатури. Останній елемент у масиві повинен містити нульовий символ `'\0'`.
3) Заповнити масиви M1 та M2 даними. Передбачити два варіанти: автоматичне заповнення довільними даними та заповнення даними, що вводяться з клавіатури.
4) Вивести на екран вміст масивів M1 та M2.
5) Створити динамічний масив M3, який містить дані масиву M1, після яких йдуть дані масиву M2. Вивести вміст масиву M3 на екран.
6) Створити динамічний рядок, який містить перші п'ять символів з масиву M1 та останні 3 символи з масиву M2.
7) За допомогою спеціально написаного метода вставити в динамічний рядок додаткові символи з M1 та M2 так, щоб його вміст збігався з вмістом масиву M3. Вивести вміст динамічного рядка на консоль.
8) Реалізувати пп. 6-7 за допомогою однозв'язного списку елементів, кожен з яких містить один символ. Вивести вміст початкового та кінцевого списку на консоль.

9) Провести звільнення всіх блоків виділеної динамічної пам’яті перед завершенням функції `main()` (до оператора `return`).

КОНТРОЛЬНІ ЗАПИТАННЯ

1. В чому полягає концепція об’єктно-орієнтованого програмування? Чим ця концепція є кращою за концепцію функціонального програмування (див. лабораторну роботу № 3)?
2. На яких принципах ґрунтується об’єктно-орієнтоване програмування?
3. Що таке клас? Чим клас відрізняється від структур у мові С? Що таке об’єкт? Чи є об’єкт синонімом класу? Наведіть приклади.
4. Чи можуть до складу класу входити елементи-дані? Елементи-функції? Наведіть приклади.
5. Що таке специфікатори доступу? Для чого вони потрібні? Як вони пов’язані з інкапсуляцією даних? Наведіть приклади.
6. Для чого в класах використовуються конструктори та деструктори? Який формат їх запису? Чим конструктор/деструктор відрізняється від «звичайного» методу класу? Наведіть приклади.
7. Чи може бути у класу кілька конструкторів? Кілька деструкторів? Якщо відповідь позитивна поясніть для чого це потрібно і наведіть приклади.
8. Чи можна визначати методи всередині оголошення класу? Зовні оголошення класу? Наведіть приклади.
9. Що таке вказівник `this`? Для чого він потрібен? Наведіть приклади.
10. Яким чином можна одержати доступ до полів класу? До його методів? Наведіть приклади.
11. Чи можна створювати масиви об’єктів класу? Наведіть приклад.
12. Що таке динамічні структури даних? Для чого вони потрібні? Які динамічні структури даних ви знаєте?
13. Поясніть принципи реалізації динамічних масивів. Чим ці масиви відрізняються від статичних масивів? В чому їх переваги і недоліки порівняно зі статичними масивами? Наведіть приклади.
14. Поясніть принципи реалізації динамічних рядків. Чим динамічні рядки відрізняються від «звичайних» текстових рядків, у чому їх переваги і недоліки? Наведіть приклади.
15. Поясніть принципи реалізації однозв’язного списку. Чим такий список відрізняється від масиву? В чому переваги і недоліки списків порівняно з масивами? Наведіть приклади.

Лабораторна робота № 9.
Успадкування. Поліморфізм.
Нові засоби мови С++. Потоки С++

Мета роботи:

1) Ознайомитись з принципами успадкування класів і навчитись за допомогою успадкування реалізовувати ієрархії класів.
2) Ознайомитись з принципами поліморфізму і навчитись реалізовувати класи з віртуальними методами.
3) Навчитись використовувати нові засоби мови С++ (простори імен, вказівники на методи, дружні класи та дружні функції, перевантаження операторів, нові правила приведення типів тощо).
4) Вдосконалити навички використання потоків С++. Навчитись застосовувати потоки С++ для виведення даних на консоль / у файл або для зчитування даних з консолі/файлу.

УСПАДКУВАННЯ

Принципи успадкування

Успадкування (англ. inheritance, іноді вживається термін *наслідування*) – це один з основних принципів об'єктно-орієнтованого програмування, який дозволяє створювати нові структуровані типи даних, що спадкують свої властивості (дані-члени) та поведінку (члени-методи) від існуючих структурованих типів даних. Успадкування може використовуватись для класів, структур, бітових полів та об'єднань. Найчастіше його застосовують саме для класів і структур.

Розглянемо нижче особливості успадкування на прикладі успадкування класів.

Для реалізації успадкування необхідно як мінімум два різні класи. Клас, на основі якого за допомогою успадкування створюється новий клас, називається ***базовим класом*** або ***батьківським класом***. Новий клас, який утворюється в результаті успадкування, називається ***похідним класом*** або ***класом-нащадком***. В процесі успадкування, тобто при утворенні зв'язку «батько – нащадок» між класами, у клас-нащадок включаються всі поля та методи батьківського класу. Завдяки цьому похідний клас за замовчуванням дублює властивості та поведінку базового класу.

Успадкування, як правило, використовується для реалізації деяких спільних рис у поведінці взаємопов'язаних класів. Для цього, зазвичай,

створюють деякий базовий клас, який реалізує бажану поведінку. Від нього породжують похідний клас, в якому ця поведінка може бути дещо змінена або уточнена. Створений похідний клас, в свою чергу, може бути базовим для наступного похідного класу, в якому поведінка класу може ще дещо змінитись, тощо. Наприклад, розглянемо деякий гіпотетичний базовий клас `CAnimal` (тварина), від якого породжується клас `CBird` (птах), який, у свою чергу, є батьківським класом для класу `CDuck` (качка). Відповідна ієрархія класів зображена на рис. 31.

Рис. 31

Клас `CAnimal` є найбільш фундаментальним класом у запропонованій ієрархії класів. В цьому класі можуть бути реалізовані властивості та поведінка, притаманні всім тваринам. Наприклад, у цьому класі можуть бути задані поля, що описують стан тварини (жива і здорова, хвора, мертва тощо), вид активності тварини (спить, їсть, рухається, відпочиває, але не спить, …), тощо. Також у цьому класі може бути реалізовано велику кількість методів, що характеризують поведінку тварини, наприклад, пошук нею їжі, порятунок від ворогів, обходження власної території тощо.

Клас `CBird` є класом, похідним від `CAnimal` і в той же час базовим класом для `CDuck`. В класі `CBird` можуть бути уточнені властивості та риси поведінки тварин, характерні саме для птахів. Наприклад, у клас `CBird` може бути додатково додана поведінка (стан) «будова гнізд», «висиджування яєць» тощо, може бути додана можливість польоту птаха.

Останній клас `CDuck` визначає властивості окремого птаха – качки. В цьому класі можуть бути передбачені спеціалізовані конструктори для задання властивостей конкретного виду качок. Також у класі можна передбачити можливість плавання та пірнання у воду для качок, додати можливість програвання аудіо-файлу з криком качки тощо.

Як видно з ієрархії класів, зображеної на рис. 31, поняття базового та похідного класів є умовним. Один і той же самий клас, наприклад, клас `CBird` на рис. 31, може бути похідним від базового класу `CAnimal`, і в той же час може сам бути базовим класом для іншого класу `CDuck`.

Нові класи, які утворюються у результаті успадкування, задовольняють тим же принципам ООП, що і базові класи. Зокрема, похідні класи, так само як і базові, слідують принципу інкапсуляції, який розглядався в лабораторній роботі № 8. Щоб забезпечити можливість доступу до членів-даних і членів-методів базового класу всередині похідного класу, і в той же час не порушувати принцип інкапсуляції, успадкування здійснюється з урахуванням т. зв. ***ключа*** або ***специфікатора доступу (успадкування)***. Цей ключ або

специфікатор доступу (успадкування) може приймати ті ж самі значення, що і специфікатор доступу до елементів класу: `public`, `protected` та `private`. Специфікатор доступу, який вживається під час успадкування, разом з специфікатором доступу, вказаним у базовому класі, спільно визначають рівень доступу до кожного елементу похідного класу. Результат дії обох цих специфікаторів доступу визначається найбільш «сильним» з обох специфікаторів – тим, який найкраще задовольняє принципу інкапсуляції. Наведена нижче таблиця демонструє всі можливі комбінації специфікаторів доступу для одержання результуючого специфікатора доступу в похідному класі (див. область виділену кольором):

Ключ доступу при успадкуванні:	**Специфікатор доступу в базовому класі:**		
	`public`	`protected`	`private`
`public`	`public`	`protected`	не доступний
`protected`	`protected`	`protected`	не доступний
`private`	`private`	`private`	не доступний

Як видно з наведеної таблиці, елемент похідного класу може бути вільно доступним (`public`) тільки, якщо він був оголошений в базовому класі з ключем доступу `public` і похідний клас утворюється з базового за допомогою ключа успадкування `public`. Ключ доступу при успадкуванні `protected` та `private` перекривають дію специфікаторів доступу `public` та `protected` для елементів базового класу. З іншого боку, елементи базового класу, які були оголошені зі специфікатором доступу `private`, виявляються недоступними у похідному класі. Звертатись до таких елементів можна лише через відкриті методи, реалізовані в базовому класі.

Залежно від значення ключа доступу при успадкуванні (ключа успадкування), іноді вживають терміни: ***відкрите успадкування*** (ключ `public`), ***захищене успадкування*** (ключ `protected`) та ***закрите успадкування*** (ключ `private`). Отже, якщо наприклад відомо, що деякий клас `B` виводиться з деякого класу `A` за допомогою захищеного успадкування, це означає, що при записі класу `B` використовується успадкування властивостей і поведінки класу `A` з ключем доступу `protected`.

Наведемо загальний список правил, за якими здійснюється успадкування:

- За замовчуванням похідний клас має доступ до всіх відкритих (`public`) та захищених (`protected`) елементів базового класу. Закриті (`private`) елементи базового класу є недоступними у похідному класі.
- За необхідності похідний клас може перевизначати рівень доступу до елементів базового класу. Це дозволяє, наприклад, отримати в похідному класі доступ до закритих елементів базового класу.
- Конструктори та деструктор ніколи не успадковуються.

- При створенні екземпляра похідного класу спочатку для цього екземпляра викликається конструктор базового класу і лише потім конструктор похідного класу.
- При знищенні об'єкта похідного класу деструктори викликаються в зворотному порядку: спочатку деструктор похідного класу, потім деструктор базового класу.
- Операція присвоювання `operator=()` не успадковується, навіть якщо вона перевантажена в базовому класі (перевантаження операцій розглядається далі).
- Вказівник на базовий клас може вказувати на будь-який похідний клас. При присвоюванні вказівнику на базовий клас адреси об'єкта похідного класу приведення типу непотрібно. Наприклад, для ієрархії класів `CAnimal`, `CBird`, `CDuck`, пов'язаних успадкуванням, наступний код є повністю коректним:

```
CAnimal  animal; // class CAnimal {};
CBird    bird;   // class CBird: public CAnimal {};
CDuck    duck;   // class CDuck: public CBird {};
CAnimal*  p = &animal;
p = &bird; // p вказує на похідний клас.
p = &duck; // p вказує на похідний клас.
CBird*    p2 = &bird;
p2 = &duck; // p2 вказує на похідний клас.
```

Просте успадкування

Просте успадкування – це успадкування, за якого у кожного похідного класу є лише *один* базовий клас. При простому успадкуванні взаємопов'язані класи формують лінійний ланцюжок, подібний до показаного на рис. 31. Похідний клас `CBird` породжується від єдиного базового класу `CAnimal`. В свою чергу клас `CDuck` успадковує свої властивості від єдиного базового класу `CBird`.

Для реалізації простого успадкування застосовується синтаксис:

```
class Базовий
{
    // Елементи базового класу.
};

class Похідний: ключ_успадкування Базовий
{
    // Елементи похідного класу.
```

```
};
```

Просте успадкування можливе лише за наявності мінімум двох класів, один з яких є базовим (батьківським), а інший – похідним (класом-нащадком). Щоб підкреслити цю особливість простого успадкування, вище оголошено деякий клас `Базовий`, який буде відігравати роль базового класу, а також клас `Похідний`, який є класом, похідним від базового класу `Базовий`. Для двох класів, `Базовий` та `Похідний`, успадкування вводиться за допомогою рядка:

```
class Похідний: ключ_успадкування Базовий
```

де `ключ_успадкування` – це одне із службових слів `public`, `protected` або `private`. Наприклад, згадана вище ієрархія класів `CAnimal`, `CBird`, `CDuck` може бути реалізована наступним чином:

```
class CAnimal
{
    // Елементи класу CAnimal.
};

class CBird: public CAnimal
{
    // Елементи класу CBird.
};

class CDuck: protected CBird
{
    // Елементи класу CDuck.
};
```

Тут спочатку вводиться клас `CAnimal`. Потім від нього породжується клас `CBird` з використанням ключа успадкування `public`. Нарешті, останній похідний клас – клас `CDuck` успадковує властивості класу `CBird` з урахуванням ключа успадкування `protected`.

У лістингу 9.1 наведено приклад використання класів, пов'язаних за допомогою простого успадкування:

Лістинг 9.1:

```
1: #include <stdlib.h>
2: #include <time.h>
3: #include <iostream>
4:
5: using namespace std;
```

```
 6:
 7: // СТАН ПОВЕДІНКИ ТВАРИНИ.
 8: #define  STATE_REST   0    // Спокій.
 9: #define  STATE_RUN    1    // Бігти.
10: #define  STATE_HUNT   2    // Полювати.
11: #define  STATE_HIDE   3    // Ховатись.
12: // Доступно для птахів:
13: #define  STATE_FLY    4    // Літати.
14: // Доступно для качок:
15: #define  STATE_SWIM   5    // Плавати.
16: #define  STATE_DIVE   6    // Пірнати.
17:
18: const char* aStateStrings[] =
19: {
20:     "Rest", "Run", "Hunt", "Hide",
21:     "Fly",
22:     "Swim", "Dive"
23: };
24:
25: class CAnimal
26: {
27: private:
28:     bool alive;
29:
30: protected:
31:     int  state;
32:
33: public:
34:     CAnimal(): alive(true), state(0) {}
35:     ~CAnimal(){}
36:     void SetRandomState()
37:     {
38:         state = rand() % (STATE_HIDE+1);
39:     }
40:     void PrintAnimalInfo( char* s )
41:     {
42:         cout << (s ? s : "Animal") << " is ";
43:         cout << (alive ? "alive. " : "dead. ");
44:         cout << "Its state is ";
45:         cout << aStateStrings[state] << endl;
46:     }
47: };
48:
49: class CBird: public CAnimal
50: {
```

```
51: public:
52:     CBird(): CAnimal() {}
53:     ~CBird(){}
54:     void SetRandomState()
55:     {
56:         state = rand() % (STATE_FLY+1);
57:     }
58:     void PrintInfo()
59:     {
60:         PrintAnimalInfo("Bird");
61:     }
62:     void Fly()
63:     {
64:         state = STATE_FLY;
65:     }
66: };
67:
68: class CDuck: public CBird
69: {
70: public:
71:     CDuck(): CBird() {}
72:     ~CDuck(){}
73:     void SetRandomState()
74:     {
75:         state = rand() % (STATE_DIVE+1);
76:     }
77:     void PrintInfo()
78:     {
79:         PrintAnimalInfo("Duck");
80:     }
81:     void Swim()
82:     {
83:         state = STATE_SWIM;
84:     }
85:     void Dive()
86:     {
87:         state = STATE_DIVE;
88:     }
89: };
90:
91: int main(void)
92: {
93:     //Ініціалізувати генератор випадкових чисел.
94:     srand( (unsigned int)time(NULL) );
95:
```

```
 96:      CAnimal animal;
 97:      CBird   bird;
 98:      CDuck   duck;
 99:
100:      animal.PrintAnimalInfo(0);
101:
102:      bird.SetRandomState();
103:      bird.PrintInfo();
104:
105:      duck.Swim();
106:      CBird* p = &duck;
107:      p->PrintAnimalInfo("Gray duck");
108:
109:      cout << endl << endl;
110:      return 0;
111:  }
```

У лістингу 9.1 вводиться базовий клас `CAnimal` (рядки 25-47), від якого шляхом простого відкритого успадкування спочатку породжується клас `CBird` (рядки 49-66), а потім від класу `CBird` породжується клас `CDuck` (рядки 68-89).

В класі `CAnimal` вводяться два відкритих метода: `SetRandomState()` та `PrintAnimalInfo()`.

Метод `SetRandomState()`, як слідує з його назви, встановлює випадковий стан поведінки тварини в межах від `STATE_REST` (значення `0`) до `STATE_HIDE` включно (рядки 36-39).

Метод `PrintAnimalInfo()` виводить інформацію про назву тварини (параметр `s`) та її стан (рядки 40-46). Якщо параметр `s` є недійсним (нульовий вказівник), в методі використовується узагальнена назва тварини – рядок `"Animal"`. Якщо ж параметр `s` є ненульовим, в методі використовується текстовий рядок, на який вказує `s`.

У класі `CBird`, який є похідним від `CAnimal`, також реалізується метод `SetRandomState()`. При цьому метод `SetRandomState()`, реалізований в класі `CBird`, *перекриває* (*перевизначає*, *ховає*) успадкований метод базового класу `CAnimal::SetRandomState()`. В цілому принцип роботи методу `CBird::SetRandomState()` ідентичний до принципу роботи методу `CAnimal::SetRandomState()`. Відмінність між ними полягає лише в різних діапазонах доступних станів (див. рядки 38 та 56). Для методу `CBird::SetRandomState()` доступним також буде стан польоту птаха, пов'язаний з макросом `STATE_FLY`.

У класі `CBird` також вводяться два нових методи. Метод `Fly()` (рядки 62-65) встановлює стан польоту для тварини (птаха). Інший метод `PrintInfo()` викликає метод базового класу `PrintAnimalInfo()` з

параметром у вигляді текстового рядка "Bird" (рядки 58-61). Реальна робота виконується саме методом CAnimal::PrintAnimalInfo(), а метод PrintInfo() відіграє роль певної «обгортки», «перехідника» для методу CAnimal::PrintAnimalInfo().

Реалізація класу CDuck схожа на реалізацію класу CBird. В класі CDuck вводяться спеціалізовані методи Swim() (рядки 81-84) та Dive() (рядки 85-88), які дозволяють задавати стан поведінки характерний саме для качки, що описується класом CDuck. Так само, як і в класі CBird, реалізується «обгортка/перехідник» – метод PrintInfo(), всередині якого викликається метод CAnimal::PrintAnimalInfo(). Також перевизначається метод SetRandomState(), в якому враховується доступність двох додаткових типів поведінки для качок: станів STATE_SWIM та STATE_DIVE.

В тілі функції main() (рядки 93-110) спочатку ініціалізується генератор випадкових чисел у рядку 94. Потім створюються три об'єкти різних класів: animal, bird та duck (рядки 96-98). За допомогою оператора animal.PrintAnimalInfo(0); виводиться інформація про тварину, що пов'язана з об'єктом animal.

У рядках 102, 103 випадковим чином змінюється стан птаха (об'єкт bird) і оновлена інформація про цей стан виводиться на консоль за допомогою методу PrintInfo() класу CBird.

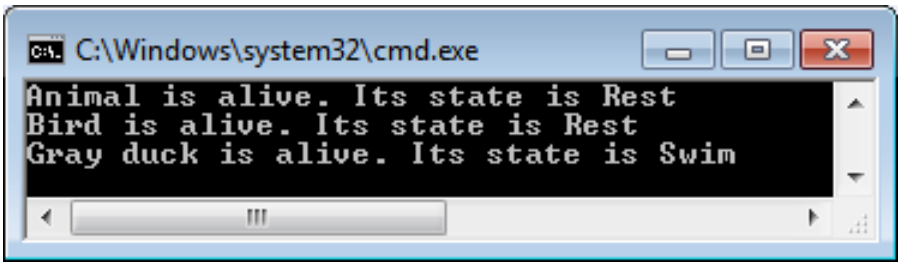


Рис. 32

В останньому блоці коду (рядки 105-107) явним чином встановлюється стан поведінки качки (об'єкт duck) шляхом виклику методу Swim(). Далі адреса об'єкту duck зберігається у вказівнику p на *базовий* клас CBird. В рядку 107 через цей вказівник викликається метод PrintAnimalInfo(), який був успадкований класом CDuck від класу CAnimal.

Результат роботи програми з лістингу 9.1 показано на рис. 32.

Перевизначення рівня доступу до елементів базового класу в похідному класі

Специфікатор доступу до елемента у базовому класі та ключ успадкування, однозначно визначають специфікатор доступу (public, protected або private) до цього елементу в похідному класі (див. табл. на стор. 276).

Проте іноді виникає необхідність змінити рівень доступу до певного окремого елемента базового класу в похідному класі. Використовувати для цього інший ключ успадкування може бути нераціонально, оскільки цей ключ буде впливати на рівень доступу до *всіх* елементів базового класу в похідному класі. Натомість бажано змінити рівень доступу лише для *певного* елементу базового класу, не змінюючи рівень доступу до інших його елементів. Для реалізації такої операції в мові C++ необхідно явно записати потрібний елемент базового класу в форматі `Ім'я_базового_класу::Ім'я_елемента` всередині похідного класу з потрібним специфікатором доступу `public`, `protected` або `private`. Наступний приклад демонструє використання цього способу перевизначення доступу:

```
class A
{
public:
    int i;
};

class B: public A
{
protected:
    A::i;    // Перевизначення доступу.
};
```

В класі `A` є єдине відкрите поле `i`. При відкритому успадкуванні класу `B` від класу `A`, поле `i`, що було введене в базовому класі `A`, залишається відкритим і в похідному класі `B`. Щоб зробити це поле захищеним, в тілі класу `B` перевизначається доступ до поля `i` базового класу `A` як `protected: A::i;`. Завдяки цьому, рядки коду

```
B b;
b.i = 3; // Помилка: доступ заборонено!
```

компілюватись не будуть, оскільки доступ до поля `i` всередині об'єкта класу `B` заборонений для зовнішнього коду. В той же час наступний фрагмент коду буде компілюватись успішно, оскільки доступ до поля `i` класу `A` є дозволеним:

```
A a;
a.i = 3; // Доступ дозволений!
```

Складне (множинне) успадкування

Створення класу на основі двох чи більше батьківських класів називається ***складним (множинним) успадкуванням***. Синтаксис утворення такого класу має наступний вигляд:

```
class A { /* ... */ };
class B { /* ... */ };
class C : ключ_A A, ключ_B B { /* ... */ };
```

де A та B – батьківські класи, а C – клас, утворений за допомогою множинного успадкування з урахуванням ключів успадкування ключ_A та ключ_B для класів A та B. Згідно наведеного прикладу, клас C буде містити члени класу A, задані з урахуванням ключа_A, і члени класу B, задані з урахуванням ключа_B.

Множинне наслідування є очевидним розширенням простого успадкування, проте при його використанні можуть траплятись ситуації з *неоднозначністю доступу* до членів похідного класу. Ця ситуація типово виникає, коли базові класи містять однойменні поля або методи. Припустимо, що існує деякий базовий клас A, в якому заданий метод f(). Далі утворимо від цього класу два похідних класи B та C за допомогою простого успадкування (рис. 33). Кожен з класів B та C буде містити метод f(), який успадкований від класу A. Тепер утворимо за допомогою множинного успадкування новий клас BC, використовуючи базові класи B та C (рис. 33). В результаті в клас BC будуть включені однойменні методи f() з обох батьківських класів B та C. Відповідно, при виклику методу f() для екземпляру класу BC виникне закономірне питання: який саме метод викликати – той, який успадкований з класу B, чи той, який успадкований з класу C? Компілятор не в змозі самостійно розібратися з цим питанням, тому при виявленні неоднозначності доступу до елементів похідного класу, він зупиняє процес компіляції і виводить повідомлення про помилку. Наступний фрагмент коду демонструє приклад виникнення такої ситуації:

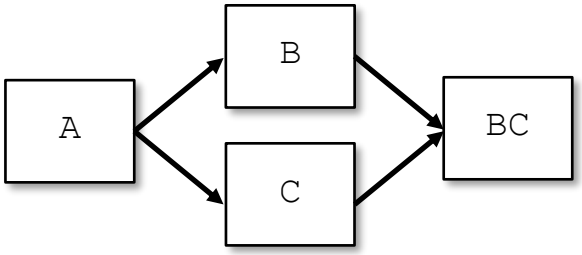


Рис. 33

```
class A
{
public:
    void f(){}
};
```

```
class B : public A {};
class C : public A {};

class BC : public B, public C {};

BC bc;
bc.f(); // Що викликати: f() з B чи з C?
```

Останній рядок коду `bc.f();` є саме тим рядком, де виникає неоднозначність доступу до методу `f()` класу `BC`, оскільки компілятор не може зрозуміти слід викликати метод `f()`, успадкований від класу `B`, чи метод `f()`, успадкований від класу `C`. Щоб прибрати цю неоднозначність, можна вказати з якого саме базового класу слід викликати метод. Це виконується за допомогою операції дозволу видимості `::`. Якщо потрібно викликати метод `f()` з класу `B`, замість рядка `bc.f();` слід написати `bc.B::f();`. Якщо ж необхідно викликати метод, успадкований від класу `C`, слід написати `bc.C::f();`.

Ще кращий спосіб уникнення появи ситуації з неоднозначністю доступу – використовувати ***віртуальні*** базові класи, що вживаються з додатковим ключем успадкування `virtual`. В цьому випадку похідний клас буде завжди містити лише єдину копію елементів віртуального базового класу. Наприклад, наступний фрагмент коду, в якому використовується віртуальний базовий клас `A`, буде компілюватись успішно:

```
class A
{
public:
    void f(){}
};

class B : virtual public A {};
class C : virtual public A {};

class BC : public B, public C {};

BC bc;
bc.f(); // BC містить єдину копію f().
```

Помилки, пов'язаної з неоднозначністю виклику методу `f()` для об'єкту `bc`, вже виникати не буде, оскільки клас `BC` містить *єдиний* варіант методу `f()` – це метод `f()`, успадкований з класу `A`.

ПОЛІМОРФІЗМ ТА ВІРТУАЛЬНІ ФУНКЦІЇ

Віртуальні функції (методи)

Віртуальна функція (метод) – це метод, який спеціально пов'язується з типом класу, завдяки чому виклик методу виявляється залежним від типу об'єкта.

Щоб зрозуміти для чого потрібні віртуальні функції розглянемо випадок, коли з деякого класу `A`, що містить метод `f()`, виводиться за допомогою успадкування клас `B`:

```
class A
{
public:
    void f(){ /* Код для класу A */ }
};

class B : public A
{
public:
    void f(){ /* Код для класу B */ }
};
```

Метод `f()` є доступним для об'єктів обох класів – `A` і `B`, проте має різну реалізацію в цих класах. Розглянемо тепер фрагмент коду:

```
A a;
a.f();
B b;
b.f();
```

В цьому фрагменті спочатку викликається метод `f()` для об'єкта `a` класу `A`, а потім однойменний метод викликається для об'єкта `b` класу `B`. Оскільки методи `f()` викликаються для об'єктів різних класів, компілятор генерує виклики різних методів (однойменних, проте реалізованих по-різному у різних класах). При цьому метод `f()` з класу `A` коректно викликається саме для об'єкта класу `A`, а метод, реалізований в класі `B`, викликається саме для об'єкта класу `B`.

Однак у наступному фрагменті коду коректність виклику метода `f()` вже порушується:

```
A* p = &b;
p->f();
```

Вказівник `p` на базовий клас `A` насправді містить адресу об'єкта `b`. При виклику `p->f();` за логікою програми мав би викликатись метод `f()` з класу `B`. Проте компілятор визначає тип об'єкта, для якого має викликатись `f()`, як `*p`, що означає об'єкт класу `A`. Тому при виклику `p->f();` помилково буде викликатись метод `f()` з класу `A`, а не однойменний метод класу `B`.

Щоб уникнути зазначеної помилки, `f()` слід оголосити *віртуальним методом* у базовому класі `A`. Тоді будь-який виклик `f()` буде відбуватись з урахуванням типу того об'єкта, для якого здійснюється виклик методу. Зокрема, при виклику `p->f();` буде спочатку визначено правильний тип об'єкту `*p` – а це клас `B`, оскільки `p = &b`, і лише потім для об'єкту `*p` буде викликано метод `f()` з класу `B`.

Таким чином, використання віртуальних функцій дозволяє більш надійно та безпечно реалізувати виклик однойменних методів у класах, пов'язаних через успадкування.

Механізм виклику віртуальних функцій базується на тому, що для кожної такої функції компілятор генерує додаткову службову інформацію (приховану від програміста). Також компілятор розміщує інформацію про віртуальні методи класу в т. зв. *таблиці віртуальних функцій*. При зверненні до об'єкту класу через вказівник або посилання[1] спочатку отримується доступ до таблиці віртуальних функцій, пов'язаної з об'єктом класу. Потім у цій таблиці знаходиться потрібний віртуальний метод і лише потім цей метод викликається. Ця технологія називається ***пізнім (динамічним) зв'язуванням***. Вона працює лише при використанні вказівників або посилань на об'єкти класу і реалізується безпосередньо в режимі реального часу – в процесі виконання програми (не під час компіляції програми[2]).

Віртуальні функції оголошуються зі спеціальним службовим словом `virtual`, яке зазначається на початку прототипу/заголовку функції. Наприклад, для класу `A`, що розглядався вище, оголошення методу `f()` як віртуального здійснюється таким чином:

```
class A
{
public:
    virtual void f(){ /* Код для класу A */ }
};
```

[1] Раніше зазначалось, що при безпосередньому використанні об'єктів класу виклик методів завжди відбувається правильно – викликаються саме методи, пов'язані з об'єктом класу.

[2] В момент, коли проводиться компіляція програми, точно визначити тип об'єкта часто буває неможливо. Тому при використанні віртуальних функцій компілятор генерує не простий виклик методу, а генерує код, який реалізує для цього методу технологію пізнього зв'язування: доступ до таблиці віртуальних функцій, доступ до потрібного методу і лише потім виклик самого методу.

При цьому метод `f()` у будь-якому класі, похідному від `A`, наприклад, у класі `B`, також буде віртуальним методом. Використовувати ще раз службове слово `virtual` для `f()` у похідному класі вже не обов'язково.

Для віртуальних функцій (методів) застосовуються наступні правила:

- Механізм віртуальних функцій може використовуватись лише при доступі до об'єкта класу через вказівник або посилання.
- Специфікатор `virtual` не є обов'язковим при перевизначенні функції в похідному класі. Достатньо оголосити функцію віртуальною в базовому класі і вона збереже свій «статус» віртуальної в будь-яких похідних класах.
- Механізм віртуальних функцій можна обійти при використанні оператора дозволу видимості `::`. Наприклад, навіть якщо метод `f()` в базовому класі `A` оголошений як віртуальний, наступний фрагмент коду призведе до блокування виклику віртуального методу `f()` з класу `B` для об'єкта `b`. Замість цього для об'єкта `b` буде викликатись метод `f()` з класу `A`:

```
B b;
A* p = &b;
p->A::f();
```

- Віртуальна функція не може бути статичною, тобто для неї неприпустима комбінація специфікаторів `virtual static`.
- Віртуальна функція має бути обов'язково визначена або вона має бути *чистою віртуальною функцією* (про це див. далі).

На практиці віртуальні функції зазвичай оголошують в базових класах. Особливо часто використовуються віртуальні деструктори, що дозволяє забезпечити надійне звільнення ресурсів об'єктів класу. Разом з тим, у конструкторах і деструкторах не рекомендується викликати віртуальні функції, оскільки такий виклик іноді може призводити до помилкової роботи програми. Наприклад, при створенні об'єкта похідного класу `B` спочатку буде викликатись конструктор базового класу `A`. Якщо в цьому конструкторі базового класу викликається віртуальний метод `f()`, реально буде здійснюватися виклик `f()` з класу `B` (оскільки створюється саме об'єкт класу `B`). Проте, в момент виконання тіла конструктора базового класу `A`, ініціалізація об'єкта класу `B` ще остаточно не виконана (конструктор `B()` не завершив свою роботу), що може призвести до появи помилок при виклику `B::f()`.

Чисті віртуальні функції та абстрактні класи

Чиста віртуальна функція (pure virtual function) – це віртуальна функція, яка *тільки оголошена* всередині класу, а її тіло має бути визначене в похідному класі. Оголошення чистої віртуальної функції має завершуватись

лексемою `=0;`. Наприклад, наступний фрагмент коду показує оголошення чистої віртуальної функції `f()` в класі `A`:

```
class A
{
public:
    virtual void f() = 0;
};
```

Клас, в якому є хоча б одна чиста віртуальна функція називається ***абстрактним*** (abstract class).

В мові C++ заборонено створювати об'єкти абстрактних класів[1], хоча можна вільно використовувати вказівники та посилання на абстрактні класи. Абстрактні класи, зазвичай, використовують лише як базові класи, функціональність яких має явним чином визначатись (уточнюватись) в похідних класах. Однак, якщо в класі, похідному від абстрактного класу, визначено не всі чисті віртуальні функції, такий похідний клас також буде абстрактним.

Особливим випадком абстрактного класу є ***чистий абстрактний клас*** (pure abstract class), який містить тільки чисті віртуальні функції і не містить полів та звичайних (не віртуальних) методів.

Абстрактні класи та чисті абстрактні класи часто використовуються як певні «класи-заготовки» або «класи-інтерфейси». В таких класах, зазвичай, оголошується одна чи кілька відкритих чистих віртуальних функцій, які мають виконувати найважливіші операції, притаманні всім класам, похідним від даного абстрактного класу. Потім від такого абстрактного класу породжують кілька похідних класів, в кожному з яких відповідні віртуальні функції визначаються явним чином. Це дає можливість задавати та контролювати характерну поведінку для всіх класів, породжених від певного (чистого) абстрактного класу. Розглянемо приклад використання віртуальних функцій та абстрактних класів, наведений у лістингу 9.2.

Лістинг 9.2:

```
1: #include <iostream>
2: #include <math.h>
3:
4: using namespace std;
5:
6: //----------------------------------------
7: // СТРУКТУРА, що описує точку на площині.
8: //----------------------------------------
9: struct CPoint
```

[1] При створенні об'єкта абстрактного класу необхідно, щоб всі методи класу були визначені, реалізовані. Ця умова не виконується, якщо в класі є чисті віртуальні функції.

```
10: {
11:     double x, y;
12:
13:     CPoint(){ Set(); }
14:     CPoint( double _x ){ Set(_x); }
15:     CPoint( double _x, double _y ){ Set(_x, _y);
16: }
17:
18:     void Set()
19:     {
20:         x = 0.0;
21:         y = 0.0;
22:     }
23:     void Set( double _x )
24:     {
25:         x = _x;
26:         y = 0.0;
27:     }
28:     void Set( double _x, double _y )
29:     {
30:         x = _x;
31:         y = _y;
32:     }
33: };
34:
35: // ЗРУЧНІ ДОПОМІЖНІ ФУНКЦІЇ.
36:
37: inline double Sqr( double x ){ return x*x; }
38:
39: double Dist( const CPoint& a, const CPoint& b )
40: {
41:     return sqrt( Sqr(a.x-b.x) + Sqr(a.y-b.y));
42: }
43:
44: //--------------------------------------
45: // Абстрактний клас для певної фігури.
46: // Не містить даних, проте регламентує
47: // формат загальних методів.
48: //--------------------------------------
49: class CShape
50: {
51: public:
52:     CShape(){}
53:     virtual ~CShape(){}// Вірт. деструктор.
54:
```

```
55: public:
56:     // Звичайні віртуальні функції-методи.
57:     virtual double Perimeter() const
58:     { return 0.0; }
59:     virtual double Area()      const
60:     { return 0.0; }
61:     // Чисті віртуальні функції-методи.
62:     virtual void   Init() = 0;
63:     virtual void   PrintInfo() const = 0;
64: };
65:
66:
67: //---------------------------
68: // Похідний клас - трикутник.
69: //---------------------------
70: class CTriangle : public CShape
71: {
72:     CPoint A, B, C;
73:
74: public:
75:     CTriangle(): CShape(){}
76:     virtual ~CTriangle(){}// Вірт. деструктор.
77:
78: public:
79:     // Перевантажені віртуальні функції-методи.
80:     virtual double Perimeter() const
81:     {
82:         double a = Dist(A, B);
83:         double b = Dist(A, C);
84:         double c = Dist(B, C);
85:
86:         return (a + b + c);
87:     }
88:     virtual double Area()      const
89:     {
90:         double a = Dist(A, B);
91:         double b = Dist(A, C);
92:         double c = Dist(B, C);
93:
94:         double p = 0.5*(a + b + c);
95:
96:         return sqrt(p*(p-a)*(p-b)*(p-c));
97:     }
98:
99:     virtual void   Init()
```

```
100:      {
101:          A.Set(1.0, 3.0); // Задати координати.
102:          B.Set(2.0, 6.0);
103:          C.Set(3.0, 7.0);
104:      }
105:      virtual void   PrintInfo() const
106:      {
107:          cout << "Triangle ABC:" << endl;
108:          cout << "    A(" << A.x;
109:          cout << " ; " << A.y << ")" << endl;
110:          cout << "    B(" << B.x;
111:          cout << " ; " << B.y << ")" << endl;
112:          cout << "    C(" << C.x;
113:          cout << " ; " << C.y << ")" << endl;
114:          cout << "Perimeter: " << Perimeter();
115:          cout << endl;
116:          cout << "Area: " << Area() << endl;
117:          cout << endl << endl;
118:      }
119: };
120:
121: //-------------------------
122: // Похідний клас - квадрат.
123: //-------------------------
124: class CRect : public CShape
125: {
126:      CPoint LeftTop, RightBottom;
127:
128: public:
129:      CRect(): CShape(){}
130:      virtual ~CRect(){}// Вірт. деструктор.
131:
132: public:
133:      // Перевантажені віртуальні функції-методи.
134:      virtual double Perimeter() const
135:      {
136:          double a = RightBottom.x - LeftTop.x;
137:          double b = RightBottom.y - LeftTop.y;
138:
139:          return 2.0*(a + b);
140:      }
141:      virtual double Area()      const
142:      {
143:          double a = RightBottom.x - LeftTop.x;
144:          double b = RightBottom.y - LeftTop.y;
```

```
145:
146:         return a * b;
147:     }
148:
149:     virtual void   Init()// Задати координати.
150:     {
151:         LeftTop.Set(1.0, 5.0);
152:         RightBottom.Set(10.0, 7.0);
153:     }
154:
155:     virtual void   PrintInfo() const
156:     {
157:         cout << "Square ABCD:" << endl;
158:         cout << "    Left-Top(" << LeftTop.x;
159:         cout << " ; " << LeftTop.y << ")";
160:         cout << endl;
161:         cout << "    Right-Bottom(";
162:         cout << RightBottom.x;
163:         cout << " ; " << RightBottom.y << ")";
164:         cout << endl;
165:         cout << "Perimeter: " << Perimeter();
166:         cout << endl;
167:         cout << "Area: " << Area() << endl;
168:         cout << endl << endl;
169:     }
170: };
171:
172:
173: int main()
174: {
175:     CShape* p1 = NULL;
176:     CRect*  p2 = NULL;
177:
178:     // Вказівник на базовий клас
179:     // вказує і на похідний клас!
180:     p1 = new CTriangle;
181:     if( !p1 ) return -1;
182:
183:     p2 = new CRect;
184:     if( !p2 )
185:     {
186:         delete p1;
187:         return -1;
188:     }
189:
```

```
190:      // Використання віртуального механізму.
191:      p1->Init();
192:      p2->Init();
193:
194:      p1->PrintInfo();
195:      p2->PrintInfo();
196:
197:      // Приклад використання поліморфізму:
198:      cout << "Area of shape 2 is " << p2->Area();
199:      cout << endl << endl;
200:      // Блокування механізму віртуальних функцій:
201:      cout << "Area of shape 2 is ";
202:      cout << p2->CShape::Area() << endl << endl;
203:
204:      if( p1 )
205:      {
206:          delete p1;
207:          p1 = NULL;
208:      }
209:      if( p2 )
210:      {
211:          delete p2;
212:          p2 = NULL;
213:      }
214:
215:      return 0;
216:  }
```

У лістингу 9.2 спочатку вводиться структура `CPoint`, що описує координати точки на площині (рядки 9-33). У рядках 37, 39-42 вводяться дві допоміжні функції: `Sqr()`, яка обчислює квадрат переданого їй аргументу, а також функція `Dist()`, яка обчислює відстань між двома точками на площині.

У рядках 49-64 вводиться базовий клас `CShape`, який описує деяку плоску фігуру. В ньому задається пустий конструктор за замовчуванням `CShape()` і пустий віртуальний деструктор `~CShape()`. Далі в класі вводяться віртуальні методи-«заглушки» `Perimeter()` та `Area()` для обчислення периметру та площі фігури. Для класу `CShape` ці методи просто повертають значення `0.0`.

У рядках 62, 63 оголошуються чисті віртуальні функції `Init()` та `PrintInfo()`, завдяки яким клас `CShape` є абстрактним класом.

Від класу `CShape` породжуються два класи-нащадки: `CTriangle` (рядки 70-119) та `CRect` (рядки 124-170).

Клас `CTriangle` описує фігуру-трикутник. Поля `A`, `B`, `C` типу `CPoint`,

введені в цьому класі, задають координати вершин трикутника. Віртуальний метод `Perimeter()` класу `CTriangle` обраховує периметр трикутника. Інший віртуальний метод `Area()` обраховує площу трикутника, використовуючи формулу Герона. При цьому вживається функція `Dist()` для знаходження довжини сторін трикутника.

В класі `CTriangle` визначаються чисті віртуальні функції, введені в базовому класі `CShape`. У методі `Init()` задаються координати вершин трикутника (рядки 99-104). Метод `PrintInfo()` виводить на консоль форматоване повідомлення з детальною інформацією про фігуру (трикутник).

Подібно до класу `CTriangle` реалізується і клас `CRect`, який описує прямокутник на площині. В класі `CRect` задаються координати лівої верхньої (`LeftTop`) та правої нижньої (`RightBottom`) вершин прямокутника.

Віртуальні методи `Perimeter()` та `Area()` обраховують периметр та площу прямокутника. В процесі такого обрахунку застосовується функція `Dist()` для знаходження довжини сторін прямокутника.

В класі `CRect` визначаються чисті віртуальні функції, введені в базовому класі `CShape`. У методі `Init()` задаються координати вершин прямокутника (рядки 149-153). Метод `PrintInfo()` виводить на консоль повідомлення з детальною інформацією про прямокутник.

В тілі функції `main()` спочатку вводяться два вказівники: `p1` та `p2`. Вказівник `p1` є вказівником на абстрактний клас `CShape`, а вказівник `p2` є вказівником на клас `CRect`. Далі, в динамічній пам'яті створюються об'єкти класів `CTriangle` та `CRect`, вказівники на які зберігаються в `p1` та `p2` (рядки 180-188).

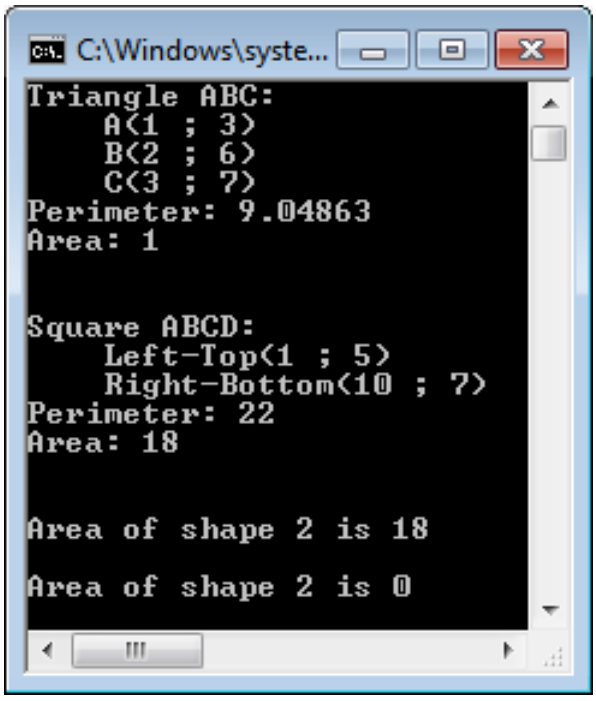


Рис. 34

Створені об'єкти ініціалізуються під час виклику віртуальних методів `Init()` (рядки 191, 192). При цьому використовується технологія пізнього зв'язування, яка дозволяє коректно ініціалізувати об'єкт класу `CTriangle`, хоча виклик методу `Init()` для цього об'єкта відбувається через вказівник на абстрактний базовий клас `CShape`. Далі інформація про ініціалізовані фігури виводиться на консоль за допомогою виклику `PrintInfo()`, під час якого знову використовується механізм віртуальних функцій.

В рядку 202 застосовується операція дозволу видимості `::`, яка дозволяє явним чином вказати, який саме

метод `Area()` має викликатись. В лістингу явно задається метод `CShape::Area()`, що належить до базового класу `CShape`. Цей метод просто повертає значення `0.0`, яке потім виводиться на консоль.

Нарешті останній блок коду (рядки 204-213) виконує звільнення динамічної пам'яті перед завершенням роботи програми.

Результат роботи програми з лістингу 9.2 показано на рис. 34.

Наприкінці зазначимо правила роботи з абстрактними класами:

- Не можна створювати об'єкти (локальні/глобальні змінні) та поля абстрактного класу.
- Абстрактний клас не може використовуватись як тип аргументу функції або як тип значення, яке вона повертає.
- Абстрактний клас не можна використовувати при явному перетворенні типу об'єкта.
- Дозволяється визначати вказівник або посилання на абстрактний клас.
- Якщо клас, похідний від абстрактного, не визначає всі чисті віртуальні функції, він також вважається абстрактним класом.

НОВІ ЗАСОБИ МОВИ С++

Мова С++ відрізняється від мови С наявністю гнучких і зручних засобів, корисних для швидкого написання складних програм. Найбільш важливими з цих засобів, безумовно, є класи, які розглядались раніше. Однак окрім класів, у мові С++ є ще цілий ряд практично важливих нововведень, деякі з яких будуть розглянуті нижче.

Простори імен та операція ::

Простір імен – це іменована область, в якій можна задавати різноманітні ідентифікатори. Всі програмні компоненти (типи, змінні, функції тощо), визначені всередині простору імен, утворюють певну логічну групу ідентифікаторів. Кожен з таких ідентифікаторів нерозривно пов'язується з іменем того простору імен, в якому він визначений.

Простори імен були введені в мову С++, щоб спростити усунення помилок, пов'язаних з конфліктом ідентифікаторів. У програмах на С/С++ досить часто виникає ситуація, коли різні програмні компоненти (типи, функції, змінні), визначені в різних частинах програми (особливо в різних бібліотеках), записуються за допомогою однакових ідентифікаторів. При цьому наявність такого «дублювання» імен призводить до появи численних помилок компіляції. Наприклад, припустимо, що в одному .c/.cpp файлі програми необхідно використати дві глобальні змінні, які мають однакове ім'я `int Error`. При

зверненні в тексті програми до однієї з цих змінних, відразу ж виникає питання, до якої саме змінної слід звернутись: до першої, чи до другої? Щоб уникнути такого конфлікту імен, звичайно, можна перейменувати одну зі змінних. Однак у мові C++ є більш елегантний спосіб усунути цей конфлікт імен – слід лише включити одну зі змінних до певного простору імен, або включити кожну з цих змінних до різних просторів імен.

Якщо при записі тексту програми простір імен не вказується, за замовчуванням вважається, що використовується ***глобальний простір імен***. Таким чином, у більшості програм, розглянутих раніше (починаючи з лабораторної роботи № 1), всі ідентифікатори, як правило, задавались в глобальному просторі імен. Винятком з цього правила були лише програми, які використовували простір імен `std`, характерний для стандартної бібліотеки C++.

Доступ до ідентифікатора `y`, визначеного всередині деякого простору імен `X` виконується за допомогою операції дозволу видимості в формі `X::y`. Якщо ім'я `X` простору імен відсутнє, операція `::y` інтерпретується як доступ до ідентифікатора `y`, визначеного всередині глобального простору імен. Наприклад, при роботі з консоллю для доступу до об'єкта потоку `cin`, визначеного всередині простору імен `std`, може використовуватись звернення `std::cin`.

Мова C++ дозволяє програмісту оголошувати власні простори імен, використовуючи службове слово `namespace`. Всі ідентифікатори, задані всередині такого простору імен, належать лише цьому простору імен. Вони стають невизначеними поза межами свого простору імен, зокрема, вони перестають бути глобальними ідентифікаторами, визначеними в межах глобального простору імен.

Наступний приклад показує створення та використання простору імен:

```
namespace NamespaceA
{
    int a;

    class A
    {
    public:
        int a;
    };
}
```

Тут у просторі імен `NamespaceA` визначена змінна `a` та клас `A`. Вони будуть доступні в основній частині програми при використанні операції дозволу видимості `::`, наприклад:

```
NamespaceA::a = 0;
```

```
NamespaceA::A b;
b.a = 1;
```

Для підключення простору імен використовується ключове слово `using`. Типовим прикладом використання `using` є рядок

```
using namespace std;
```

Цей рядок вказує компілятору на необхідність розпізнавання та врахування всіх ідентифікаторів, визначених в межах вказаного простору імен `std`. Запис цього рядка на початку програми дозволяє спростити використання коду стандартної бібліотеки C++.

Наприкінці зазначимо, що простори імен можуть бути вкладеними. Наприклад, нехай деяка змінна `x` визначена як

```
namespace A
{
    namespace B
    {
        int x;
    }
}
```

Щоб звернутись до неї всередині функції `main()` слід використати програмну конструкцію `A::B::x`.

Вказівники на методи класу

Методи класу є функціями. Отже, вказівник на метод класу по суті є вказівником на функцію (див. лабораторну роботу № 5). Синтаксис визначення такого вказівника наступний:

```
Тип (*Клас::Ім'я_вказівника)(Аргументи) = Значення;
```

Тут `Тип` – це тип результату, який має повертати метод.

`Ім'я_вказівника` – це ім'я *типу вказівника*, пов'язаного з потрібним методом, визначеним всередині класу `Клас`. Метод повинен мати перелік аргументів `Аргументи` і повертати величину вказаного `Типу`. Ім'я типу вказівника має записуватись із зірочкою перед ним і має обов'язково включатись в круглі дужки.

`Значення`, яким може ініціалізуватись вказівник на метод, є іменем методу.

При виклику будь-якого нестатичного метода йому завжди передається прихований вказівник this. Отже, для виклику методу через вказівник слід використовувати програмні конструкції, в яких явно зазначається об'єкт, для якого викликається метод, та Конкретні_аргументи, що передаються методу:

```
(Об'єкт.*Ім'я_вказівника)(Конкретні_аргументи);
(*this.*Ім'я_вказівника)(Конкретні_аргументи);
(this->*Ім'я_вказівника)(Конкретні_аргументи);
(АдресаОб'єкта->*Ім'я_вказівника)(Конкретні_аргументи);
(*АдресаОб'єкта.*Ім'я_вказівника)(Конкретні_аргументи);
```

В цих прикладах Об'єкт є об'єктом класу, в якому існує метод, на який вказує вказівник Ім'я_вказівника. АдресаОб'єкта є адресою відповідного об'єкта в пам'яті, тобто значенням вказівника &Об'єкт.

Перший спосіб виклику метода за допомогою конструкції Об'єкт.*Ім'я_вказівника може використовуватись лише для конкретного відомого Об'єкту (із відомим іменем). Останні два способи не потребують знання імені Об'єкту, достатньо лише мати вказівник на потрібний об'єкт. Другий та третій способи виклику, де використовується вказівник this, можна застосовувати тільки всередині об'єкта Класу, в якому існує метод, на який вказує вказівник Ім'я_вказівника.

Розглянемо приклад використання вказівників на методи (лістинг 9.3).

Лістинг 9.3:

```
 1: #include <stdio.h>
 2:
 3: class A
 4: {
 5:     int i;
 6:
 7: public:
 8:     A() : i(0) {}
 9:     A( int _i ) : i(_i) {}
10:
11:     void Change( int _i ){ i = _i; }
12:
13:     int Print()
14:     {
15:         printf("A::i == %d\n", i);
16:         return i;
17:     }
18:
```

```
19:      void CallMethodPtr( int (A::*methodPtr)() )
20:      {
21:           (*this.*methodPtr)();
22:           (this->*methodPtr)();
23:      }
24: };
25:
26: int main()
27: {
28:      int (A::*methodPtr)() = &A::Print;
29:
30:      A a(10);
31:      (a.*methodPtr)();
32:
33:      a.Change(20);
34:      a.CallMethodPtr(methodPtr);
35:
36:      A* pA = &a;
37:      pA->Change(30);
38:      (pA->*methodPtr)();
39:
40:      pA->Change(40);
41:      ((*pA).*methodPtr)();
42:
43:      return 0;
44: }
```

У лістингу 9.3 спочатку вводиться клас `A`, в якому є закрите поле `int i`, два конструктори та три методи: `Change()`, `Print()` та `CallMethodPtr()`.

Метод `Change()` змінює вміст поля `i` (див. рядок 11).

Метод `Print()` виводить на консоль вміст поля `i` і потім повертає значення цього поля (рядки 13-17).

Останній метод `CallMethodPtr()` викликає метод класу `A`, вказівник на який передається параметром (рядки 19-23). Параметр методу `CallMethodPtr()` має тип `int (A::*methodPtr)()`. Цей параметр є вказівником на деякий метод класу `A`, який немає аргументів і повертає величину типу `int`. Єдиним методом класу `A`, який задовольняє цьому критерію, є метод `Print()`.

В тілі методу `CallMethodPtr()` виконуються два оператори: `(*this.*methodPtr)();` та `(this->*methodPtr)();`. Обидва вони зводяться до виклику методу, вказівник на який передавався параметром у `CallMethodPtr()`. В першому випадку виклик реалізується за допомогою

розіменування вказівника `this` та використання операції «крапка» `.`. В другому випадку до вказівника `this` застосовується операція «стрілка» `->`. Власне сам виклик методу здійснюється шляхом розіменування вказівника `*methodPtr` і використання круглих дужок `()` для виклику методу.

В тілі функції `main()` спочатку ініціалізується вказівник на метод класу `A` за допомогою рядка 28: `int (A::*methodPtr)() = &A::Print;`. Вказівник `methodPtr` вказує на метод `Print()`, що визначений в класі `A`.

В рядках 30-31 створюється локальний ініціалізований об'єкт `a` класу `A` і для цього об'єкта через вказівник `methodPtr` викликається метод `Print()`. Виклик здійснюється оператором `(a.*methodPtr)();`.

У наступному блоці коду (рядки 33-34) вміст єдиного поля об'єкту `a` змінюється, після чого викликається метод `CallMethodPtr()`. Всередині методу `CallMethodPtr()` двічі через вказівник `methodPtr` викликається метод `Print()`.

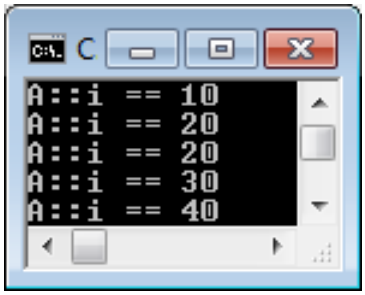


Рис. 35

В рядках 36-38 вводиться вказівник `pA` на об'єкт `a` класу `A`. Через цей вказівник `pA` змінюється вміст об'єкта `a` (рядок 37). Потім викликається метод `Print()` для об'єкта `a` за допомогою конструкції `(pA->*methodPtr)();`.

Останній блок коду змінює поле `i` всередині об'єкта `a`, що адресується вказівником `pA` (рядок 40). Далі, у рядку 41, для об'єкта, що пов'язаний з `*pA`, знову викликається метод `Print()` за допомогою конструкції `(pA->*methodPtr)();`.

Результат роботи програми з лістингу 9.3 показано на рис. 35.

Дружні функції та дружні класи

Специфікатори доступу `protected` та `private` забороняють доступ до елементів класу для функцій, визначених поза межами відповідного класу. Проте іноді виникає потреба, щоб деяка функція, визначена поза межами класу, мала повний доступ до усіх його елементів. В мові C++ є вбудований механізм, який дозволяє це реалізувати – це оголошення потрібної функції як *дружньої* до відповідного класу.

Дружня (`friend`) ***функція*** – це функція, яка не є елементом класу (вона визначена поза його межами), але яка має повний доступ до усіх елементів класу (зокрема, до його закритих та захищених елементів). Для того щоб вказати, що деяка функція є дружньою до певного класу, її прототип слід написати в оголошенні класу і перед прототипом вказати службове слово `friend`. Приклад оголошення та використання дружньої функції наведено в лістингу

9.4.

Лістинг 9.4:

```
 1: class A
 2: {
 3:     int n;
 4: public:
 5:     A(){ n = 0; }
 6:     void Add( int b ){ n = n+b; }
 7:     // Оголошення дружньої функції.
 8:     friend void friendFunction( A &x );
 9: };
10:
11: void friendFunction( A &x )
12: {
13:     x.n=0; // Функція має повний доступ до x.n.
14: }
15:
16: int main()
17: {
18:     A a;                  // a.n = 0.
19:     a.Add(1);             // a.n += 1.
20:     friendFunction(a);    // a.n = 0.
21:     return 0;
22: }
```

У рядках 1-9 оголошується клас `A`, в якому зазначається (рядок 8), що функція `friendFunction()` є дружньою до класу `A`. На це вказує модифікатор `friend`, записаний перед прототипом функції `friendFunction()`.

Оскільки функція `friendFunction()` оголошена дружньою до класу `A`, вона має повний доступ до всіх елементів класу `A`. Тому код `x.n=0;` в тілі цієї функції буде успішно компілюватись і виконуватись, хоча поле `n` в класі `A` є закритим.

Рядок 20 в тілі функції `main()`, `friendFunction(a);`, демонструє виклик дружньої функції.

Дружні функції не входять до області дії класу по відношенню до якого вони оголошені дружніми. Це означає, що такі функції не можуть викликатись для об'єкта відповідного класу за допомогою операцій «крапка» `.` чи «стрілка» `->`. Однак, якщо дружня функція сама є методом іншого класу, вона може викликатись як звичайний метод за допомогою операцій `.` або `->` для об'єктів «свого» класу, в якому вона визначена. Наприклад, якщо в деякому класі `B` є метод `int f(A&, int)`, цей метод можна зробити дружнім до класу `A`, додавши рядок `friend int B::f(A&, int);` до оголошення класу `A`.

Виклик цього дружнього методу може бути реалізованим наступним чином:

```
A a;
B b;
b.f(a, 10);
```

Окрім функцій, один ***клас*** можна зробити ***дружнім*** до іншого. Для цього в оголошення деякого класу `A` слід додати рядок `friend class B;`. Це дозволяє «відкрити» всім членам *дружнього класу* `B` доступ до всіх елементів класу `A`. Однак цей спосіб автоматично не працює в зворотному напрямку. Тобто, хоча клас `B` має повний доступ до елементів класу `A`, клас `A` немає повного доступу до елементів класу `B`. Якщо потрібно, щоб клас `A` також мав повний доступ до елементів класу `B`, у клас `B` слід додати рядок `friend class A;`. Відповідний фрагмент коду для двох взаємно дружніх класів `A` та `B` може виглядати так:

```
class A, B; // Неповне визначення типу.
class A
{
    // ...
    friend class B;
};
class B
{
    // ...
    friend class A;
};
```

Спочатку використовується неповне визначення типу, яке просто інформує компілятор, що є два класи: `A` та `B`. Наповнення цих класів при цьому поки що не деталізується. Потім явним чином оголошується клас `A`, в якому зазначається, що клас `B` є дружнім. Останнім оголошується клас `B`, в якому вказується, що клас `A` є дружнім.

Дружні функції та дружні класи підкоряються наступним правилам:

- Дружня функція або дружній клас ігнорують будь-які специфікатори доступу у тому класі, по відношенню до якого вони оголошені дружніми.
- Функція може бути оголошена дружньою одночасно для кількох класів.
- Дія модифікатора `friend` невзаємна: якщо клас `A` оголошує клас `B` дружнім, це не означає, що клас `A` є дружнім до `B`.
- Дружність не успадковується: якщо клас `A` оголошує клас `B` дружнім, класи, похідні від `B`, не будуть автоматично вважатись дружніми по відношенню до класу `A`.
- Дружність не зберігається при успадкуванні: якщо клас `A` оголошує клас `B`

дружнім, класи, похідні від A, не будуть автоматично визнавати дружнім клас B.

Перевантаження операцій (операторів)

Операції, які є елементами мови C++ (див. лабораторну роботу № 1), реалізуються в C++ за допомогою спеціальних функцій-операторів. Ці функції-оператори мають наперед визначені імена у вигляді `operatorXXX`, де замість `XXX` вказується позначення операції, прийняте в мові програмування C/C++. Наприклад, вираз типу `a + b`, можна інтерпретувати як виклик функції-оператора `operator+(a, b)`.

Різні функції-оператори, які вміють працювати з даними стандартних типів (`char`, `int`, `double` тощо), вбудовані у мову C++. Проте, за необхідності, програміст може самостійно реалізувати більшість таких функцій-операторів і для нестандартних типів даних, введених самим програмістом. Це дозволяє писати програми, в яких вживаються доволі прості та зрозумілі конструкції типу `c = a + b;`, для змінних `a`, `b`, `c` довільного типу. Для цього необхідно ***перевантажити*** потрібну операцію (функцію-оператор) – тобто «навчити» її працювати з даними конкретного нестандартного типу.

Перевантаження операцій (операторів) здійснюється за тими ж правилами, що і перевантаження функцій (див. лабораторну роботу № 3). Однак при перевантаженні операцій слід враховувати і деякі специфічні обмеження, притаманні саме операціям у мовах C/C++:

- Перевантаження можна реалізовувати лише для деяких операцій, вбудованих у мову C++. Введення нових операцій (наприклад, `**`, `@`) не дозволяється.
- Дозволяється перевантажувати тільки наступні операції:

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`
`~`	`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`
`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`
`--`	`->`	`->*`	`()`	`[]`	`,`	`new`	`delete`
`(тип)`							

- Операції `?:`, `.`, `.*`, `::`, `sizeof`, `typeid`, операції препроцесора `#` і `##` та нові операції приведення типу (див. нижче) перевантажувати не можна.
- Перевантажений оператор не може змінити поведінку операції для стандартних типів даних.
- Перевантаження операції не впливає на її пріоритет, асоціативність та кількість операндів.
- Перевантажений оператор має бути або членом класу, або має приймати

принаймні один аргумент, що пов'язаний з класом, для якого перевантажується операція.

- Перевантажені оператори не можуть мати аргументів за замовчуванням.
- За виключенням `operator=()`, перевантажені оператори не успадковуються.

Одним з найпростіших способів перевантаження операторів є використання дружніх функцій, введених з ключовим словом `friend`. Приклад реалізації перевантажених операцій з використанням дружніх функцій і методів класу `CComplex`, що описує комплексне число, наведений в лістингу 9.5.

Лістинг 9.5:

```
 1: #include <math.h>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: // Простий клас для комплексних чисел.
 7: class CComplex
 8: {
 9:     // За замовчуванням private:
10:     double re;
11:     double im;
12:
13: public:
14:     // Конструктори.
15:     CComplex(): re(0.0), im(0.0) {}
16:     CComplex( double _re )
17:     {
18:         re = _re;
19:         im = 0.0;
20:     }
21:     CComplex( double _re, double _im )
22:     {
23:         re = _re;
24:         im = _im;
25:     }
26:     CComplex( const CComplex& r )
27:     {
28:         re = r.re;
29:         im = r.im;
30:     }
31:     // Деструктор.
32:     ~CComplex(){}
33:
```

```
34:     // Доступ до даних.
35:     double Re() const { return re; }
36:     double Im() const { return im; }
37:
38:     // Зміна даних.
39:     void Re( double _re ) { re = _re; }
40:     void Im( double _im ) { im = _im; }
41:
42:     // Інші методи.
43:     double Module() const
44:     {
45:         return sqrt(re*re + im*im);
46:     }
47:     double Arg() const
48:     {
49:         return atan2(im, re);
50:     }
51:
52: public:
53:     // Перевантажені оператори -- методи.
54:     CComplex operator+=( double a )
55:     {
56:         re += a;
57:         return *this;
58:     }
59:     CComplex operator+=( const CComplex& a )
60:     {
61:         re += a.re;
62:         im += a.im;
63:         return *this;
64:     }
65:     CComplex operator-=( double a )
66:     {
67:         re -= a;
68:         return *this;
69:     }
70:     CComplex operator-=( const CComplex& a )
71:     {
72:         re -= a.re;
73:         im -= a.im;
74:         return *this;
75:     }
76:
77: public:
78:     // Перевантажені дружні (friend) оператори.
```

```
 79:     friend CComplex operator+(
 80:         const CComplex& a, const CComplex& b );
 81:     friend CComplex operator+( double a, const
 82:         CComplex& b );
 83:     friend CComplex operator+(
 84:         const CComplex& a, double b );
 85:
 86:     friend ostream& operator<<( ostream& os,
 87:         CComplex& r );
 88: };
 89:
 90:
 91: CComplex operator+( const CComplex& a, const
 92: CComplex& b )
 93: {
 94:     return CComplex(
 95:         a.Re()+b.Re(), a.Im()+b.Im());
 96: }
 97:
 98: CComplex operator+( double a, const CComplex& b )
 99: {
100:     return CComplex(a+b.Re(), b.Im());
101: }
102:
103: CComplex operator+( const CComplex& a, double b )
104: {
105:     return CComplex(a.Re()+b, a.Im());
106: }
107:
108: ostream& operator<<( ostream& os, CComplex& r )
109: {
110:     os << "CComplex(" << r.Re() << "; ";
111:     os << r.Im() << ")";
112:     return os;
113: }
114:
115: int main(void)
116: {
117:     CComplex c1(2.5, -3.6);
118:     cout << c1 << endl;
119:
120:     CComplex c2(-0.4, 1.65);
121:     cout << c2 << endl;
122:
123:     cout << "c1 + c2 = ";
```

```
124:
125:      CComplex c3 = c1 + c2;
126:      cout << c3 << endl;
127:
128:      cout << "Module   of " << c3 << " is ";
129:      cout << c3.Module() << endl;
130:      cout << "Argument of " << c3 << " is ";
131:      cout << c3.Arg() << endl << endl << endl;
132:
133:      return 0;
134:  }
```

У лістингу 9.5 оголошується клас `CComplex`, який вже розглядався раніше в лабораторній роботі № 8. Цей клас призначений для зберігання інформації про комплексне число (поля `re` та `im`) та реалізації різних операцій з комплексними числами. Важливими нововведеннями у класі `CComplex` є перевантажені оператори `operator+=()`, `operator-=()`, `operator+()`, `operator<<()`.

Перевантажені оператори `operator+=()` та `operator-=()` введені як методи класу `CComplex`. Реалізація операторів `operator+=()` та `operator-=()` є дуже подібною, тому далі детально розглянемо реалізацію тільки `operator+=()`.

Перевантажена операція `operator+=()` визначена в двох варіантах: коли до об'єкта класу `CComplex` додається дійсне число типу `double` (рядки 54-58) та коли до об'єкта додається комплексне число типу `CComplex` (рядки 59-64).

Оператор `operator+=( double a )` спочатку додає до дійсної частини комплексного числа (поля `re` об'єкту) переданий аргумент `a` (рядок 56). Після цього вже модифікований об'єкт класу `CComplex` повертається за допомогою оператора `return *this;`.

Схожим чином працює і оператор `CComplex operator+=( const CComplex& a )`. Спочатку в тілі цього оператора модифікуються поля `re` та `im` об'єкта класу (рядки 61, 62). Потім модифікований об'єкт повертається за допомогою оператора `return *this;`.

Перевантажені оператори `operator+()`, `operator<<()` визначені як дружні функції для класу `CComplex` (рядки 79-87). Реалізація операторів `operator+()`, що мають різні параметри, майже ідентична. Розглянемо, наприклад, реалізацію оператора `CComplex operator+( const CComplex& a, const CComplex& b )`. Весь код цієї функції-оператора складається лише з єдиного оператора `return`, який повертає результат додавання двох комплексних чисел, заданих через сталі посилання `a` та `b` (рядки 94, 95). Результат операції задається як безіменний об'єкт, що утворюється при

виклику конструктора `CComplex( double _re, double _im )`. Перший аргумент конструктора `CComplex` задається як сума дійсних частин переданих комплексних чисел `a.Re()+b.Re()`, а другий аргумент – як сума уявних частин `a.Im()+b.Im()`. Зазначимо, що для доступу до дійсної та уявної частин комплексного числа використовуються відкриті методи `Re()` та `Im()`. Такий спосіб доступу до полів `re`, `im` є дозволеним завжди, навіть, за умови, що оператор `CComplex operator+( const CComplex&, const CComplex& )` був би оголошеним без службового слова `friend`. Але оскільки перевантажений оператор `operator+()` визначений як дружній по відношенню до класу `CComplex`, в тілі цього оператора можна безпосередньо звертатись до полів `re` та `im`. Наприклад, відповідний код можна було б записати так:

```
return CComplex(a.re+b.re, a.im+b.im);
```

Останній перевантажений оператор, пов'язаний з класом `CComplex`, – це ***оператор виведення в потік*** `ostream& operator<<( ostream& os, CComplex& r )`. Цей оператор формально описує операцію побітового зсуву `<<`, яка в мові C/C++ застосовується лише для цілих чисел (або до величин, які можуть бути інтерпретовані як цілі числа). Для комплексних чисел операція побітового зсуву `<<` не має сенсу, проте позначення цієї операції `<<` можна використати для реалізації іншої операції з комплексним числом. Типово операцію `<<` для нестандартних типів реалізують як операцію виведення даних у потік C++, подібно до того, як це реалізовано для стандартних типів мови C++ (див. лабораторну роботу № 1). Саме за таким принципом реалізовано і операцію `<<` для класу `CComplex`.

Оператор `ostream& operator<<( ostream& os, CComplex& r )`, записаний в рядках 108-113, має два параметри. Перший параметр – це посилання `os` на потік типу `ostream`, в який будуть виводитись дані. Другий параметр – це посилання `r` на об'єкт класу `CComplex`, дані якого мають виводитись в потік. В тілі перевантаженого оператора послідовно виводяться дані про вміст об'єкта класу `CComplex` (рядки 110, 111). Після цього оператор `return os;` повертає посилання на потік. За рахунок того, що оператор `operator<<()` повертає посилання на потік, з'являється можливість писати рядки коду, в яких послідовно використовується операція `<<` для виводу даних в потік. Наприклад, рядки подібні до `os << "CComplex(" << r.Re() << "; ";`.

В тілі функції `main()` створюються три об'єкти класу `CComplex`. Вміст об'єктів `c1` та `c2` задається явним чином (рядки 117, 120). Їх вміст виводиться на консоль за допомогою операції `<<`, що приводить до використання перевантаженого оператора `operator<<( ostream& os, CComplex&`

r).

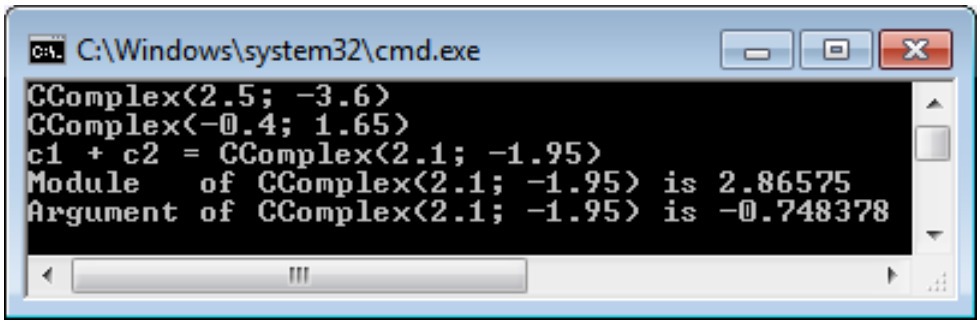


Рис. 36

Об'єкт `c3` класу `CComplex` задається як сума об'єктів `c1` та `c2` (рядок 125). Вміст цього об'єкту, а також інформація про модуль та аргумент відповідного комплексного числа виводяться на консоль через об'єкт потоку `cout` (рядки 126, 128-131).

Результат роботи програми з лістингу 9.5 показано на рис. 36.

Нові правила приведення типів

Мова С реалізує єдиний спосіб приведення деякого `виразу` (змінної, константи тощо) до потрібного `типу`: `(тип)вираз` або `тип(вираз)` або `(тип)(вираз)`. Цей спосіб приведення типу діє і в мові С++. Проте, на відміну від мови С, у С++ часто трапляється ситуація, коли тип деякого об'єкта або вказівника на об'єкт є невідомим на момент компіляції програми. Цей тип однозначно визначається лише під час роботи програми. Така ситуація є типовою для класів, пов'язаних через успадкування. Наприклад, нехай `A` – деякий базовий клас, який містить віртуальний метод `f()`, тоді клас `B`, який є похідним від `A`, також буде містити метод `f()`. В такій ситуації наступний фрагмент програми може сприйматись неоднозначно:

```
A a;
B b;
A* p = /* Адреса &a або &b */;
p->f();
```

Оскільки вказівник `p` може вказувати як на об'єкт класу `A`, так і на об'єкт класу `B`, перш ніж викликати віртуальний метод `f()`, компілятор повинен визначити на який саме об'єкт вказує `p`: на `a` чи `b`? Результат цієї операції на момент компіляції програми визначеним бути не може, тому тип об'єкта, на який вказує `p`, визначається лише під час роботи програми. Отже, в мові С++ існує потреба в реалізації спеціальної операції визначення/приведення типу,

яка могла б виконуватись в режимі реального часу, впродовж усього часу роботи програми.

В мові C++ реалізуються чотири нові способи приведення певного виразу v до деякого типу T:

```
static_cast<T>(v)
dynamic_cast<T>(v)
const_cast<T>(v)
reinterpret_cast<T>(v)
```

Операція приведення типу `static_cast<T>(v)` по суті нічим не відрізняється від операції приведення типу в мові C. Такий оператор добре працює при застосуванні до неполіморфних типів (що не містять віртуальних функцій і не пов'язані через успадкування).

Операція `dynamic_cast<T>(v)` може застосовуватись для об'єктів довільного типу, зокрема, й для класів, що містять віртуальні методи та пов'язані між собою через успадкування. Якщо приведення типу можливе на момент компіляції програми, `dynamic_cast<T>(v)` виконує ті ж самі дії, що й `static_cast<T>(v)`.

Операція `const_cast<T>(v)` використовується для видалення у виразу v атрибутів `const` або `volatile`[1].

Нарешті, остання операція `reinterpret_cast<T>(v)` змінює точку зору компілятора на тип величини v, проте не змінює двійкове представлення v у пам'яті. По суті, ця операція зводиться до «переосмислення», «зміни» інтерпретації для заданого набору бітів, що відповідають v.

Приклад використання нових правил приведення типів демонструється в лістингу 9.6.

Лістинг 9.6:

```
 1: class A { /* ... */ };
 2: class B: public A { /* ... */ };
 3: class C: public B { /* ... */ };
 4:
 5: typedef unsigned __int64 UINT64;
 6:
 7: int main(void)
 8: {
 9:     C c;
10:     C* pc = &c;
11:     A* pa = static_cast<A*>(pc);
```

[1] Атрибут `volatile` означає, що вміст змінної може змінюватись ззовні програми (іншою програмою, операційною системою, драйвером периферійного пристрою тощо). Компілятор не оптимізує роботу зі змінними, які оголошені з атрибутом `volatile`.

```
12:      B* pb = dynamic_cast<B*>(pc);
13:      const C* pcc = &c;
14:      C* p = const_cast<C*>(pcc);
15:      UINT64 u = reinterpret_cast<UINT64>(pc);
16:
17:      return 0;
18:  }
```

В лістингу 9.6 вводяться три класи: базовий клас `A`, похідний від нього клас `B` і похідний від `B` клас `C`. Також для спрощення вводиться новий тип `UINT64`, який є 64-бітним цілим числом без знаку.

В тілі функції `main()` створюється об'єкт `c` класу `C` і визначається вказівник `pc` на об'єкт `c` (рядки 9, 10). Далі у рядку 11 виконується статичне приведення типу за допомогою оператора `A* pa = static_cast<A*>(pc);`.

Динамічне приведення типу здійснюється у наступному рядку 12: `B* pb = dynamic_cast<B*>(pc);`. Воно буде успішним, оскільки клас `C` є похідним від `B`, отже, вказівник `pc` може бути конвертований у тип `B*`.

У рядках 13, 14 спочатку вводиться сталий вказівник `const C* pcc`, атрибут `const` якого далі прибирається за рахунок використання приведення типу `const_cast<C*>(pcc)`.

Останній рядок 15 реалізує перетворення вказівника `pc` у 64-бітне беззнакове ціле число.

Потоки C++

Потоки в стандартній бібліотеці C++

Частиною стандартної бібліотеки C++ є бібліотека потоків C++, яка включає в себе кілька класів для введення/виведення даних на консоль та у файли. На відміну від функцій стандартної бібліотеки C, потоки C++ ґрунтуються на використанні технології перевантаження операцій і тому є більш надійними при роботі з даними різного типу. Крім того, потоки C++ можуть бути легко адаптовані для роботи з будь-якими типами даних і в той же час забезпечують однаковий інтерфейс.

Доступ до бібліотеки потоків C++ забезпечується при підключенні до програми файлів заголовків `iostream`, `fstream` та ряду інших.

В програмах на C++, зазвичай, використовуються п'ять класів потоків: `std::istream`, `std::ostream`, `std::ifstream`, `std::ofstream`, `std::istream`, `std::fstream`, або класи, похідні від вказаних. Всі ці класи задані в просторі імен `std`. Клас `istream` реалізує потік введення

даних (input stream), а клас `ostream` – потік виведення даних (output stream). Класи `ifstream`, `ofstream` та `fstream` призначені для роботи з ***файловими потоками*** – потоками C++, які використовуються для введення/виведення даних у файли. Клас `ifstream` (input file stream) використовується для зчитування даних з файлу, а клас `ofstream` (output file stream) – для запису даних у файл. Останній клас `fstream` підтримує як операції зчитування даних з файлу, так і операції запису даних у файл.

Незалежно від виду потоку (звичайний чи файловий), операція запису даних в потік здійснюється за допомогою перевантаженого оператора `>>`, а операція зчитування даних з потоку – за допомогою перевантаженого оператора `<<`. Приклад використання цих перевантажених операцій розглядався в лабораторній роботі № 1.

Оскільки кожен з перевантажених операторів `>>`, `<<` для потоків повертає посилання на об'єкт потоку, перевантажені оператори одного виду (тільки `>>` або `<<`) можна записувати послідовно, в один рядок. При цьому виникають т. зв. «зчеплені» оператори, наприклад, `cout << a << b << c << d;`, де `a`, `b`, `c`, `d` – деякі змінні/об'єкти, для яких реалізована операція виведення в потік `<<`.

Розширення потоків для роботи з даними нестандартного типу

Потоки C++ можна «навчити» працювати з даними будь-якого типу. Для цього достатньо перевантажити оператори `>>` та `<<` для потрібного `Типу` даних. Відповідні оператори мають наступний формат:

```
// Зчитування даних з потоку.
std::istream& operator>>( std::istream& is, Тип& x );
// Запис даних у потік.
std::ostream& operator<<( std::ostream& is, Тип& x );
```

Тут `x` – посилання на об'єкт заданого `Типу`, вміст якого має виводитись у потік `std::ostream`, або зчитуватись з потоку `std::istream`. Приклад реалізації перевантаженого оператора `<<` для типу `CComplex` наведено в лістингу 9.5.

Робота з текстовими файлами за допомогою потоків

За замовчуванням дані зчитуються з файлових потоків і записуються в файлові потоки у текстовій формі, так само як це відбувається при роботі з консоллю. Тому для роботи з текстовими файлами, слід лише створити

потрібний об’єкт файлового потоку і далі для нього викликати перевантажені оператори (>> та/або <<). Для закриття раніше відкритого файлу слід скористатись методом `close()` для об’єкта потоку. Лістинг 9.7 демонструє типові операції, потрібні для запису/зчитування даних з текстового файлу.

Лістинг 9.7:

```
 1: #include <iostream>
 2: #include <fstream>
 3: #include <string>
 4:
 5: using namespace std;
 6:
 7: int main(void)
 8: {
 9:     string name;
10:     string read;
11:
12:     cout << "Please write your name: ";
13:     cin >> name;
14:
15:     // Відкрити файловий потік.
16:     ofstream ofs("Lab09CPP.txt", ios::out);
17:     // Записати дані в файл.
18:     ofs << name;
19:     // Закрити файловий потік.
20:     ofs.close();
21:
22:     // Відкрити файловий потік.
23:     ifstream ifs("Lab09CPP.txt", ios::in);
24:     // Зчитати дані з файлу.
25:     ifs >> read;
26:     // Закрити файловий потік.
27:     ifs.close();
28:
29:     // Вивести результати на консоль.
30:     cout << endl << "Your name is: ";
31:     cout << read << endl << endl;
32:
33:     return 0;
34: }
```

В програмі використовуються стандартні об’єкти потоків `cin`, `cout`, які визначаються у файлі <iostream>, стандартні рядки `string` (файл заголовку <string>) та файлові потоки, що стають доступними при підключенні

файлу <fstream>.

Після того, як користувач введе своє ім'я (рядки 12-13), у рядку 16 створюється об'єкт ofs файлового потоку ofstream. Цей об'єкт пов'язується з файлом "Lab09CPP.txt", який відкривається для запису даних (прапорець ios::out). За допомогою оператора ofs << name; введене ім'я, яке збережене в об'єкті name, виводиться в потік ofs, після чого потік закривається при виклику ofs.close();.

Далі файл "Lab09CPP.txt" відкривається знову, тепер вже для читання. Це відбувається при створенні об'єкта ifs класу ifstream (рядок 23). З об'єкта потоку зчитуються дані в стандартний рядок read, після чого потік закривається при виклику ifs.close();. На завершення функції main() інформація про зчитане з файлу "Lab09CPP.txt" ім'я (read) виводиться на консоль (рядки 30-31).

Робота з бінарними файлами за допомогою потоків

Перевантажені оператори >> та << виконують введення/виведення інформації лише в текстовій формі, отже, вони не підходять для роботи з бінарними файлами, де дані повинні зберігатись в «сирому» (бінарному) форматі. Тому для роботи з бінарними файлами замість операторів >> та <<, використовують методи read() та write(), реалізовані для класів потоків ifstream та ofstream. Принцип роботи та синтаксис виклику цих методів нагадує принцип роботи та синтаксис для функцій fread() та fwrite() зі стандартної бібліотеки C, які розглядались в лабораторній роботі № 6.

Лістинг 9.8 демонструє типові операції, потрібні для запису/зчитування даних з бінарного файлу.

Лістинг 9.8:

```
 1: #include <iostream>
 2: #include <fstream>
 3:
 4: using namespace std;
 5:
 6: int main(void)
 7: {
 8:     float f1, f2;
 9:
10:     cout << "Please input float number: ";
11:     cin >> f1;
12:
13:     // Відкрити файловий потік.
```

```
14:     ofstream ofs("Lab09CPP.bin", ios::binary);
15:     // Записати дані в файл.
16:     ofs.write((char*)&f1, sizeof(f1));
17:     // Закрити файловий потік.
18:     ofs.close();
19:
20:     // Відкрити файловий потік.
21:     ifstream ifs("Lab09CPP.bin", ios::binary);
22:     // Зчитати дані з файлу.
23:     ifs.read((char*)&f2, sizeof(f2));
24:     // Закрити файловий потік.
25:     ifs.close();
26:
27:     // Вивести результати на консоль.
28:     cout << endl << "You entered: " << f1;
29:     cout << endl << "From file: " << f2;
30:     cout << endl << endl;
31:
32:     return 0;
33: }
```

Розглянемо лише відмінності програми з лістингу 9.8 від аналогічної програми з лістингу 9.7.

У рядках 10-11 з консолі зчитується число типу `float`. У рядку 16 створюється об'єкт `ofs` файлового потоку `ofstream`, який пов'язується з бінарним файлом `"Lab09CPP.bin"`, на що вказує прапорець-параметр `ios::binary`. Дані `f1` записуються в файл за допомогою виклику `ofs.write((char*)&f1, sizeof(f1));`. Першим параметром `write()` вказується адреса початку блоку даних (вказівник типу `char*`), які слід вивести у файл. Другий параметр визначає розмір блоку даних у байтах. Метод `write()` по суті виводить `sizeof(f1)` байт даних як набір з символів, що починається з адреси `&f1`. Після цього потік закривається при виклику `ofs.close();`.

Далі файл `"Lab09CPP.bin"` відкривається знову, тепер вже для читання. Це відбувається при створенні з прапорцем `ios:: binary` об'єкта `ifs` класу `ifstream` (рядок 21). З об'єкта потоку зчитуються дані за допомогою виклику `ifs.read((char*)&f2, sizeof(f2));`, після чого потік закривається при виклику `ifs.close();`.

На завершення функції `main()` інформація про введені користувачем дані `f1` та зчитані з файлу `"Lab09CPP.bin"` дані `f2` виводяться на консоль (рядки 28-30).

Форматування виводу даних. Маніпулятори потоків

Для потоків C++ передбачено три способи керування форматуванням даних, які виводяться в потік. Це безпосередній виклик *методів для форматування даних*, використання *прапорців форматування* та застосування *маніпуляторів*.

У наступній таблиці наведено стислий опис найбільш поширених методів для форматування виводу даних в потік:

Метод:	Опис:
`fill()` `fill(char)`	Повертає поточне значення символу заповнення для потоку або змінює цей символ.
`precision()` `precision(int)`	Повертає поточне значення точності дійсних чисел (кількості ненульових цифр), яке буде використовуватись при виводі даних в потік, або змінює це значення.
`width()` `width(int)`	Повертає поточне значення ширини поля виводу для потоку або змінює це значення.

З кожним потоком C++ асоціюється значна кількість прапорців форматування, що визначають в якому саме форматі інформація має виводитись у потік і зчитуватись з потоку. В наступній таблиці наведено стислий опис найбільш поширених прапорців форматування, введених в класі `ios`, який є базовим для всіх класів потоків:

Прапорець:	Опис:
`ios::dec`	Якщо встановлений, цілі числа виводяться як числа по основі 10. Встановлений за замовчуванням. Не сумісний з `ios::oct` та `ios::hex`.
`ios::fixed`	Якщо встановлений, дійсні числа виводяться з фіксованою кількістю цифр після десяткової крапки – у форматі ±nnn.dddd.
`ios::hex`	Якщо встановлений, цілі числа виводяться як числа по основі 16. Не сумісний з `ios::oct` та `ios::dec`.
`ios::internal`	Якщо встановлений, при виведенні числа, знак числа вирівнюється по лівій межі поля, а саме число – по правій межі. Проміжок між знаком та числом заповнюється поточним заповнюючим символом. Не сумісний з `ios::left` та `ios::right`.
`ios::left`	Якщо встановлений, дані, що виводяться, вирівнюються по лівій межі поля виводу. Не сумісний з `ios::internal` та `ios::right`.
`ios::oct`	Якщо встановлений, цілі числа виводяться як числа

Прапорець:	Опис:
	по основі 8. Не сумісний з `ios::hex` та `ios::dec`.
`ios::right`	Якщо встановлений, дані, що виводяться, вирівнюються по правій межі поля виводу. Не сумісний з `ios::internal` та `ios::left`.
`ios::scientific`	Якщо встановлений, дійсні числа виводяться в науковій нотації – у форматі ±n.ddddE±yy.
`ios::showbase`	Якщо встановлений при виведенні цілих чисел вказується індикатор основи (0 для чисел по основі 8 та 0x для чисел по основі 16).
`ios::showpoint`	Якщо встановлений, при виведенні дійсних чисел обов'язково виводиться десяткова крапка.
`ios::showpos`	Якщо встановлений, для додатних величин виводиться знак +.
`ios::skipws`	Якщо встановлений, під час введення даних ігноруються пробіли або еквівалентні їм символи. За замовчуванням встановлений.
`ios::unitbuf`	Якщо встановлений, внутрішній буфер потоку очищується після кожної операції запису в потік.
`ios::uppercase`	Якщо встановлений, символи від A до F та знак експоненти E в представленні чисел виводяться великими літерами.

Для роботи з прапорцями форматування використовуються спеціальні методи `flags()`, `flags(long)`, `setf(long)`, `unsetf(long)` тощо. Метод `flags()` повертає, а метод `flags(long)` встановлює значення прапорців форматування для об'єкта потоку. Метод `setf(long)` встановлює, а метод `unsetf(long)` скидає вказані прапорці форматування для об'єкта потоку.

Маніпулятори – це спеціальні функції, які можна включати в ланцюжок послідовних операцій запису `<<` та зчитування `>>` даних з потоків. Маніпулятори дозволяють проводити «зчіплювання» маніпуляторів як між собою, так і з операторами `<<` та `>>`. В наступній таблиці наведено стислий опис найбільш поширених маніпуляторів:

Маніпулятор:	Опис:
`dec`	Встановлює основу 10 при виводі цілих чисел.
`endl`	Поміщає у вихідний потік символ переходу на новий рядок `'\n'` і потім викликає маніпулятор `flush`.
`ends`	Поміщає у вихідний потік нульовий символ (символ кінця рядка) `'\0'`.
`flush`	Примусово виводить всі дані, що знаходяться в

Маніпулятор:	Опис:
	буфері потоку (на консоль, у файл тощо).
hex	Встановлює основу 16 при виводі цілих чисел.
oct	Встановлює основу 8 при виводі цілих чисел.
setbase(int b)	Задає основу при введенні/виведенні цілих чисел. Якщо параметр b дорівнює 0 або 10 числа сприймаються як числа по основі 10. Якщо b дорівнює 8 або 16 числа сприймаються як числа у вісімковій або шістнадцятковій системі числення.
setfill(int f)	Задає заповнюючий символ f для потоку.
setw(int w)	Задає значення внутрішньої ширини поля при введенні/виведенні даних у потік.
ws	Дозволяє ігнорувати при введенні даних пробіли або аналогічні їм символи.

У лістингу 9.9 наведено приклади використання усіх трьох способів форматування даних при роботі зі стандартними об'єктами потоків cin та cout.

Лістинг 9.9:

```
 1: #include <iostream>
 2: #include <iomanip>
 3:
 4: using namespace std;
 5:
 6: int main(void)
 7: {
 8:     float f;
 9:     int   i;
10:
11:     cout << "Please input float number: ";
12:     cin >> f;
13:
14:     // Встановити точність виводу чисел.
15:     cout.precision(4);
16:     // Встановити заповнюючий символ.
17:     cout.fill('.');
18:     // Встановити ширину поля.
19:     cout.width(20);
20:     // Вивести результат.
21:     cout << "You entered: " << f;
22:     cout << endl << endl;
23:
24:     // Скинути прапорець fixed.
```

```
25:       cout.unsetf(ios::fixed);
26:       // Встановити прапорець scientific.
27:       cout.unsetf(ios::scientific);
28:       // Вивести результат.
29:       cout << "In scientific form: " << f;
30:       cout << endl << endl;
31:
32:       // Знайти цілу частину числа 100*f.
33:       i = (int)(f*100.0f);
34:
35:       // Вивести результати на консоль.
36:       cout << setfill('_');
37:       cout << setw(0);
38:       cout << "100 * number =" << endl;
39:       cout << setw(8);
40:       cout << "Dec: " << dec << i << endl;
41:       cout << setw(10);
42:       cout << "Oct: " << oct << i << endl;
43:       cout << setw(12);
44:       cout << "Hex: " << hex << i << endl;
45:       cout << endl << endl;
46:
47:       return 0;
48:  }
```

Для використання засобів форматування введенням/виведенням даних у потоки до програми має бути підключеним файл заголовку `<iomanip>`.

В тілі функції `main()` спочатку зчитується з консолі число типу `float` (рядки 11, 12). Далі за допомогою методів `precision()`, `fill()` та `width()` для об'єкта `cout` встановлюються кількість цифр, які будуть виводитись для дійсних чисел (4), заповнюючий символ (`'.'`) та ширина поля виводу (20). Після цього введене дійсне число виводиться на консоль (рядки 21-22).

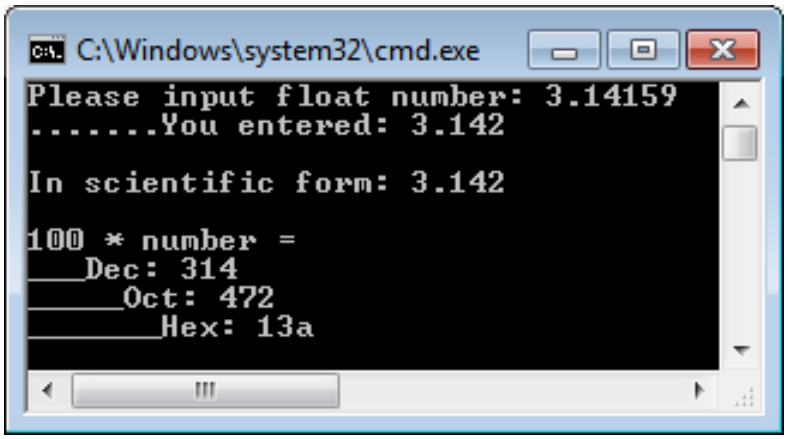


Рис. 37

У наступному блоці коду (рядки 25-30), скидається прапорець `ios::fixed` і встановлюється прапорець `ios::scientific`. Після цього введене дійсне число знову виводиться на консоль (рядки 29-30).

У рядку 33 розраховується ціла частина

від величини 100×f, де f – число введене користувачем. Далі за допомогою виклику маніпулятора `setfill('_')` змінюється заповнюючий символ для потоку. Потім здійснюється виведення розрахованого цілого числа в різних системах числення: десятковій (маніпулятор `dec`), вісімковій (маніпулятор `oct`) та шістнадцятковій (маніпулятор `hex`). Перед кожним виведенням числа явно задається ширина поля виведення за допомогою маніпулятора `setw()`.

Результати роботи програми з лістингу 9.9 для введеного числа 3.14159 показано на рис. 37.

Завдання

Варіант 1

1) Створити клас з ім'ям `Student`. В даному класі створити поля, в яких міститься ПІБ студента/студентки, його/її рік народження, курс, на якому вчиться студент/студентка, та оцінки, отримані ним/нею за сесію (5 оцінок у вигляді масиву з 5-ти елементів). Задати конструктор та деструктор класу.
2) В класі `Student` реалізувати функцію-метод, яка визначає середню оцінку студента/студентки за сесію.
3) За допомогою успадкування на основі класу `Student` утворити клас-нащадок `AppliedPhysicsStudent`, в якому додати поле з оцінками за лабораторні роботи (3 оцінки за 3 різні курси лабораторних робіт – у вигляді масиву з 3-х елементів).
4) Перевантажити функцію розрахунку середньої оцінки студента/студентки за сесію так, щоб вона також враховувала і оцінки за лабораторні роботи. Реалізувати функцію таким чином, щоб вона коректно розраховувала середню оцінку за сесію, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити дані для п'яти студентів (5 об'єктів класу `AppliedPhysicsStudent`). Оцінки для кожного студента генеруються випадковим чином за допомогою функції `rand()` з інтервалу 60-100 балів.
6) Реалізувати оператор виведення в потік `<<` даних про студента/студентку.
7) Створити текстовий файл з назвою Lab09.txt та перенести в нього за допомогою перевантаженого оператора `<<` наявну інформацію про студентів/студенток: ПІБ, рік народження, курс, середня оцінка. Порядок розташування записів в текстовому файлі має відповідати зростанню середньої оцінки. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 2

1) Створити клас з ім'ям `Auto`, який містить інформацію про автомобіль. В даному класі створити поля, в яких містяться назва (марка) машини, об'єм баку з паливом (від 50 до 100 літрів) та базова витрата палива на одиницю шляху (від 5 л до 10 л на 100 км шляху). Задати конструктор та деструктор класу.
2) В класі `Auto` реалізувати функцію-метод, яка визначає відстань (у км), яку може проїхати машина без додаткової заправки пальним.
3) За допомогою успадкування на основі класу `Auto` утворити два класи-нащадки: легковий автомобіль (`Car`) та вантажівка (`Truck`). В класі `Car` додати поле, де буде вказана кількість пасажирів (від 0 до 5), а в класі `Truck` – поле, де буде вказана вага вантажу (від 0 до 10 тон).
4) В обох класах перевантажити функцію розрахунку відстані, яку може проїхати автомобіль без дозаправки, вважаючи, що кожен пасажир збільшує базову витрату пального на 5%, а кожна тонна ваги – на 10%. Реалізувати функцію таким чином, щоб вона коректно проводила розрахунок, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити даними 3 об'єкти класу `Car` та 3 об'єкти класу `Truck`. Назва та кількість пасажирів чи вага вантажу мають задаватись явним чином при створенні об'єкта, об'єм пального у баку та базова витрата пального генеруються випадковим чином за допомогою функції `rand()` так, щоб вони потрапляли у вказані в п. 1 інтервали.
6) Реалізувати оператор виведення в потік `<<` даних про автомобіль.
7) Створити текстовий файл з назвою Lab09.txt та перенести в нього інформацію з усіх об'єктів класів `Car` та `Truck`. Порядок розташування записів в текстовому файлі має відповідати пройденій машиною відстані – від найменшої до найбільшої. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 3

1) Створити клас з ім'ям `Student`. В даному класі створити поля, в яких міститься ПІБ студента/студентки, його/її рік народження, курс, на якому вчиться студент/студентка, та оцінки, отримані ним/нею за сесію (4 оцінки у вигляді масиву з 4-х елементів). Задати конструктор та деструктор класу.
2) В класі `Student` реалізувати функцію-метод, яка визначає середню оцінку студента/студентки за сесію.
3) За допомогою успадкування на основі класу `Student` утворити клас-нащадок `AppliedPhysicsStudent`, в якому додати поле з оцінками за лабораторні роботи (4 оцінки за 4 різні курси лабораторних робіт – у вигляді масиву з 4-х елементів).

4) Перевантажити функцію розрахунку середньої оцінки студента/студентки за сесію так, щоб вона також враховувала і оцінки за лабораторні роботи. Реалізувати функцію таким чином, щоб вона коректно розраховувала середню оцінку за сесію, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити дані для чотирьох студентів (4 об'єкти класу `AppliedPhysicsStudent`). Оцінки для кожного студента генеруються випадковим чином за допомогою функції `rand()` з інтервалу 40-100 балів.
6) Реалізувати оператор виведення в потік `<<` даних про студента/студентку.
7) Створити текстовий файл з назвою Lab09.txt та перенести в нього за допомогою перевантаженого оператора `<<` наявну інформацію про студентів/студенток: ПІБ, рік народження, курс, середня оцінка. Порядок розташування записів в текстовому файлі має відповідати зменшенню середньої оцінки. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 4

1) Створити клас з ім'ям `Auto`, який містить інформацію про автомобіль. В даному класі створити поля, в яких містяться назва (марка) машини, об'єм баку з пальним (від 40 до 80 літрів) та базова витрата пального на одиницю шляху (від 4 л до 9 л на 100 км шляху). Задати конструктор та деструктор класу.
2) В класі `Auto` реалізувати функцію-метод, яка визначає відстань (у км), яку може проїхати машина без додаткової заправки пальним.
3) За допомогою успадкування на основі класу `Auto` утворити два класи-нащадки: легковий автомобіль (`Car`) та вантажівка (`Truck`). В класі `Car` додати поле, де буде вказана кількість пасажирів (від 0 до 4), а в класі `Truck` – поле, де буде вказана вага вантажу (від 0 до 5 тон).
4) В обох класах перевантажити функцію розрахунку відстані, яку може проїхати автомобіль без дозаправки, вважаючи, що кожен пасажир збільшує базову витрату пального на 10%, а кожна тонна ваги – на 5%. Реалізувати функцію таким чином, щоб вона коректно проводила розрахунок, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити даними 3 об'єкти класу `Car` та 3 об'єкти класу `Truck`. Назва та кількість пасажирів чи вага вантажу мають задаватись явним чином при створенні об'єкта, об'єм палива у баку та базова витрата палива генеруються випадковим чином за допомогою функції `rand()` так, щоб вони потрапляли у вказані в п. 1 інтервали.
6) Реалізувати оператор виведення в потік `<<` даних про автомобіль.
7) Створити текстовий файл з назвою Lab09.txt та перенести в нього

інформацію з усіх об'єктів класів `Car` та `Truck`. Порядок розташування записів в текстовому файлі має відповідати пройденій машиною відстані – від найбільшої до найменшої. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 5

1) Створити клас з ім'ям `Student`. В даному класі створити поля, в яких міститься ПІБ студента/студентки, його/її рік народження, курс, на якому вчиться студент/студентка, та оцінки, отримані ним/нею за сесію (5 оцінки у вигляді масиву з 5-ти елементів). Задати конструктор та деструктор класу.
2) В класі `Student` реалізувати функцію-метод, яка визначає середню оцінку студента/студентки за сесію.
3) За допомогою успадкування на основі класу `Student` утворити клас-нащадок `AppliedPhysicsStudent`, в якому додати поле з оцінками за лабораторні роботи (2 оцінки за 2 різні курси лабораторних робіт – у вигляді 2-х окремих полів).
4) Перевантажити функцію розрахунку середньої оцінки студента/студентки за сесію так, щоб вона також враховувала і оцінки за лабораторні роботи. Реалізувати функцію таким чином, щоб вона коректно розраховувала середню оцінку за сесію, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити дані для шести студентів (шість об'єктів класу `AppliedPhysicsStudent`). Оцінки для кожного студента генеруються випадковим чином за допомогою функції `rand()` з інтервалу 0-100 балів.
6) Реалізувати оператор виведення в потік `<<` даних про студента/студентку.
7) Створити текстовий файл з назвою `Lab09.txt` та перенести в нього за допомогою перевантаженого оператора `<<` наявну інформацію про студентів/студенток: ПІБ, рік народження, курс, середня оцінка. Порядок розташування записів в текстовому файлі має відповідати прізвищам студентів/студенток, записаним за абеткою. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 6

1) Створити клас з ім'ям `Resistor1`. Він описує резистор у вигляді відрізка дроту *квадратного* поперечного перерізу. В даному класі створити поля, в яких міститься назва металу, з якого виготовлено резистор, його довжина (в метрах, від 0 м до 1 м), поперечний розмір – сторона квадрату (від 0,001 м до 0,01 м) та питомий опір металу. Задати конструктор та

деструктор класу.

2) В класі `Resistor1` реалізувати функцію-метод, яка визначає опір резистора за даними, які містяться в полях класу.
3) За допомогою успадкування на основі класу `Resistor1` утворити клас-нащадок `Resistor2`, який описує резистор у вигляді дроту *круглого* поперечного перерізу. В цьому випадку поперечний розмір резистора має інтерпретуватись як діаметр дроту.
4) Перевантажити для класу `Resistor2` метод розрахунку опору резистора, враховуючи форму поперечного перерізу дроту. Реалізувати функцію таким чином, щоб вона коректно проводила розрахунок для резистора з різною (квадратною чи круглою) формою поперечного перерізу, навіть, коли вона викликається через вказівник на базовий клас (використати механізм віртуальних функцій).
5) Створити і заповнити дані для 5 об'єктів класу `Resistor1` та 5 об'єктів класу `Resistor2`. Довжина та поперечний розмір резистора генеруються випадковим чином за допомогою функції `rand()` так, щоб вони потрапляли у вказані в п. 1 інтервали. Матеріал випадковим чином обирається серед наступних металів: срібло, мідь, алюміній, золото, залізо. Значення питомого опору відповідних металів взяти з довідкової літератури.
6) Реалізувати оператор виведення в потік `<<` даних про резистор.
7) Створити текстовий файл з назвою `Lab09.txt` та перенести в нього інформацію з усіх об'єктів класів `Resistor1` та `Resistor2`. Порядок розташування записів в текстовому файлі має відповідати зміні опору резистора – від найменшого до найбільшого. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 7

1) Створити клас з ім'ям `Person`. В даному класі створити поля в яких міститься ПІБ людини та дата її народження. Задати конструктор та деструктор класу.
2) В класі `Person` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури дані про людину та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить відповідні дані на консоль.
3) За допомогою успадкування на основі класу `Person` створити клас-нащадок `Student`, в якому додати поле з оцінками за сесію (4 оцінки в інтервалі від 0 до 100) та номер курсу, на якому вчиться студент/студентка.
4) В класі `Student` реалізувати метод, який розраховує середню оцінку студента/студентки за сесію.
5) В класі `Student` реалізувати метод, який визначає розмір стипендії за наступними правилами: якщо є хоча б одна оцінка менше 60,

студент/студентка не отримує стипендію; якщо середній бал від 60 до 70, студент/студентка отримує базову стипендію (100%), якщо від 70 до 80 – 110% від базової стипендії, якщо від 80 до 90 – 120% від базової стипендії, якщо від 90 до 100 – 130% від базової стипендії.

6) Перевантажити метод `Input()`, який має запитувати оцінки за сесію та метод `Output()`, який додатково має друкувати середню оцінку за сесію та розмір стипендії.
7) Реалізувати оператор виведення в потік `<<` даних про студента/студентку.
8) Створити і заповнити даними 2 об'єкти класу `Student`.
9) Створити текстовий файл з назвою Lab09.txt та заповнити у нього наступну інформацію про студентів/студенток: ПІБ, рік народження, курс, середня оцінка, розмір стипендії. Порядок розташування записів в текстовому файлі має бути наступним: спочатку студенти/студентки, які отримують стипендію, за абеткою, потім ті, хто не отримують, теж за абеткою. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 8

1) Створити клас `Point`, який описує точку на площині з декартовими координатами *x*, *y*. Поля, які містять координати, повинні мати специфікатор доступу `private`. Задати конструктор та деструктор класу.
2) В класі `Point` реалізувати віртуальні методи для встановлення заданих координат точки та виведення цих координат на консоль. Реалізувати віртуальний метод `Init()`, який задає координати точки за допомогою функції `rand()` як випадкові числа в межах від –10 до 10.
3) За допомогою успадкування на основі класу `Point` створити клас-нащадок `Point3D`, що описує точку в тривимірному просторі, декартова координата *z* якої є закритим полем.
4) Створити дружні функції `Distance()`, які дозволяють розрахувати відстань між точками на площині та точками в тривимірному просторі.
5) Створити в динамічній пам'яті пару об'єктів типу `Point` та пару об'єктів типу `Point3D`. Заповнити їх даними, використовуючи метод `Init()`, і розрахувати відстань між точками.
6) Реалізувати оператор виведення в потік `<<` даних про точку.
7) Створити текстовий файл з назвою Lab09.txt та вивести в нього наступну інформацію про точки: тип точки, координати точок відповідного типу, відстань між точками. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 9

1) Створити клас `Polynom3`, який описує поліном третього порядку.

Закриті поля даного класу повинні містити коефіцієнти відповідного поліному, а також символьне позначення змінної (наприклад, 'x', або 'y'). Задати конструктор та деструктор класу. В конструкторі за замовчуванням Polynom3() коефіцієнти полінома мають задаватись нульовими, а символьне позначення змінної має задаватись як 'x'.

2) В класі Polynom3 реалізувати віртуальний метод Input(), який зчитує з клавіатури або зберігає передані дані про поліном (коефіцієнти та позначення змінної) і записує їх у поля класу, а також віртуальний метод Output(), який виводить на консоль поліном у вигляді формули, на кшталт 3x^3 + 8x^2 – 6x + 3.
3) Для класу Polynom3 реалізувати віртуальний метод, який обчислює похідну від вказаного поліному і виводить її на екран у вигляді формули, на кшталт 9x^2 + 16x – 6. Для збереження та виведення інформації про похідну скористатись функціоналом розробленого класу Polynom3.
4) За допомогою успадкування на основі класу Polynom3 створити клас-нащадок Polynom4, який описує поліном 4 порядку. В клас Polynom4 додати ще одне поле для збереження коефіцієнта при старшому ступені.
5) Для класу Polynom3 реалізувати віртуальний метод, який обчислює інтеграл від вказаного поліному і виводить його на екран у вигляді формули. Для збереження та виведення інформації про інтеграл від поліному скористатись функціоналом розробленого класу Polynom4.
6) Реалізувати оператор виведення в потік << даних про поліноми.
7) Створити 3 об’єкти класу Polynom3 і заповнити їх даними.
8) Створити текстовий файл з назвою Lab09.txt та вивести в нього у вигляді формул самі поліноми, пов’язані з об’єктами класу Polynom3, похідні та інтеграли від цих поліномів. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 10

1) Створити клас Polynom2, який описує поліном другого порядку. Захищені поля даного класу повинні містити коефіцієнти відповідного поліному, а також символьне позначення змінної (наприклад, 'x', або 'y'). Задати конструктор та деструктор класу. В конструкторі за замовчуванням Polynom2() коефіцієнти полінома мають задаватись нульовими, а символьне позначення змінної має задаватись як 'x'.
2) В класі Polynom2 реалізувати віртуальний метод Input(), який зчитує з клавіатури або зберігає передані дані про поліном (коефіцієнти та позначення змінної) та записує їх у поля класу, а також віртуальний метод Output(), який виводить на консоль поліном у вигляді формули, на кшталт 8x^2 – 6x + 3.
3) За допомогою успадкування на основі класу Polynom2 створити клас-

нащадок Polynom4, який описує поліном 4 порядку. В клас Polynom4 додати ще два захищених поля для збереження коефіцієнтів при старших степенях змінної.

4) Для класу Polynom4 реалізувати віртуальний метод, який обчислює другу похідну від вказаного поліному і виводить її на екран у вигляді формули, на кшталт 16x – 2. Для збереження та виведення інформації про подвійну похідну скористатись функціоналом розробленого класу Polynom2.
5) Для класу Polynom2 реалізувати віртуальний метод, який обчислює подвійний інтеграл для вказаного поліному і виводить його на екран у вигляді формули, на кшталт 3x^4 – 2x^3 – 3x – 7. Для збереження та виведення інформації про подвійний інтеграл скористатись функціоналом розробленого класу Polynom4.
6) Реалізувати оператор виведення в потік << даних про поліноми.
7) Створити 3 об'єкти класу Polynom4 і заповнити їх даними.
8) Створити текстовий файл з назвою Lab09.txt та вивести в нього у вигляді формул самі поліноми, пов'язані з об'єктами класу Polynom4, другі похідні від цих поліномів, а також подвійні інтеграли від розрахованих подвійних похідних. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 11

1) Створити клас з ім'ям BachelorStudent. В даному класі створити захищені поля, в яких міститься ПІБ студента/студентки, його/її рік народження, рік вступу до університету та середня оцінка за весь час навчання в бакалавраті. Задати конструктор та деструктор класу.
2) В класі BachelorStudent реалізувати віртуальний метод, який виводить інформацію про студента/студентку в форматованому вигляді.
3) За допомогою успадкування на основі класу BachelorStudent створити клас-нащадок MasterStudent, в якому додати ще два поля: рік вступу до магістратури та середню оцінку за час навчання в магістратурі.
4) Перевантажити в класі MasterStudent функцію виводу інформації про студента/студентку так, щоб при виведенні враховувались і дані за період навчання в магістратурі.
5) Реалізувати оператор виведення в потік << даних про студента/студентку.
6) Створити і заповнити дані для п'яти студентів (5 об'єктів класу MasterStudent). ПІБ та дата народження студента/студентки мають вводитись автоматично або з клавіатури. Роки вступу генеруються випадковим чином за допомогою функції rand() в діапазоні 2015–2025, а випадкові значення середніх оцінок генеруються в інтервалі 60–100 балів.
7) Створити текстовий файл з назвою Lab09.txt та перенести в нього за допомогою перевантаженого оператора << наявну інформацію про

студентів/студенток. Порядок розташування записів в текстовому файлі має відповідати року вступу в магістратуру: від меншого до більшого. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 12

1) Створити клас з ім'ям `Person`. В даному класі створити поля в яких міститься ПІБ людини та дата її народження. Задати конструктор та деструктор класу.
2) В класі `Person` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури або зберігає передані дані про людину та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить відповідні дані на консоль.
3) За допомогою успадкування на основі класу `Person` створити клас-нащадок `Worker` (співробітник), в якому додати поле `Position` (посада), яке може набувати трьох значень: `Junior`, `Middle` та `Senior`, а також поля `Experience`, що містить стаж роботи в роках, та `Salary`, де вказано базову зарплату.
4) В класі `Worker` реалізувати метод, який визначає розмір заробітної плати за наступними правилами: кожен рік стажу збільшує базову зарплату на 100 одиниць, після чого у випадку посади `Middle` платня збільшується ще на 30%, а `Senior` – на 50%.
5) Перевантажити метод `Input()`, який має додатково запитувати у користувача посаду співробітника та генерувати випадковим чином поле `Salary` за допомогою функції `rand()` з інтервалу 1000–5000 одиниць. Перевантажити метод `Output()`, який додатково має виводити інформацію про посаду, стаж та розмір базової заробітної плати.
6) Реалізувати оператор виведення в потік `<<` даних про співробітника.
7) Створити і заповнити даними 5 об'єктів класу `Worker`.
8) Створити текстовий файл з назвою `Lab09.txt` та вивести в нього наступну інформацію про співробітника: ПІБ, дата народження, посада, стаж, розмір базової зарплатні та розмір повної зарплатні (з урахуванням стажу та посади). Записи в текстовому файлі мають розташуватись в порядку зменшення повної зарплати (від найбільшої – до найменшої). Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 13

1) Створити клас з ім'ям `Book`. В даному класі створити поля, в яких міститься ПІБ автора книги, її назва, рік видання та кількість сторінок. Задати конструктор та деструктор класу.

2) В класі `Book` реалізувати віртуальний метод, який виводить на екран інформацію про книгу у форматованому вигляді.
3) За допомогою успадкування на основі класу `Book` створити клас-нащадок `LibraryBook`, в якому додати наступні поля: ПІБ людини, яка взяла книгу, та дату (день, місяць та рік), коли книга була взята.
4) Перевантажити метод виводу інформації про книгу в класі `LibraryBook` так, щоб цей метод виводив інформацію про книгу та те, хто і коли її взяв.
5) Додати в клас `LibraryBook` новий метод, який рахуватиме кількість днів, що минули з моменту видачі книги до вказаної дати.
6) Реалізувати оператор виведення в потік `<<` даних про книгу.
7) Створити дружню функцію, яка буде виводити інформацію про всі книжки певного автора. Створити дружню функцію, яка буде виводити інформацію про всі книжки, взяті певним користувачем бібліотеки.
8) Створити і заповнити даними 5 об'єктів класу `LibraryBook`.
9) Створити текстовий файл з назвою Lab09.txt та вивести в нього всю наявну інформацію про книжки, пов'язані з об'єктами класу `LibraryBook`. Порядок розташування записів в текстовому файлі має відповідати кількості днів, що минули з моменту видачі книжки (у порядку зменшення). Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 14

1) Створити клас з ім'ям `Person`. В даному класі створити поля в яких міститься ПІБ людини та дата її народження. Задати конструктор та деструктор класу.
2) В класі `Person` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури або зберігає передані дані про людину та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить відповідні дані на консоль.
3) За допомогою успадкування на основі класу `Person` створити клас-нащадок `Worker` (співробітник), в якому додати поле `Position` (посада), яке може набувати трьох значень: `Junior`, `Middle` та `Senior`, а також поля `Experience`, що містить стаж роботи в роках, та `Salary`, де вказано базову зарплату.
4) В класі `Worker` реалізувати метод, який визначає розмір заробітної плати за наступними правилами: кожен рік стажу збільшує базову зарплату на 100 одиниць, після чого у випадку посади `Middle` платня збільшується ще на 30%, а `Senior` – на 50%.
5) Перевантажити метод `Input()`, який має додатково запитувати у користувача посаду співробітника та генерувати випадковим чином поле `Salary` за допомогою функції `rand()` з інтервалу 1000–5000 одиниць.

Перевантажити метод `Output()`, який додатково має виводити інформацію про посаду, стаж та розмір базової заробітної плати.

6) Реалізувати оператор виведення в потік `<<` даних про співробітника.
7) Створити і заповнити даними 5 об'єктів класу `Worker`.
8) Створити текстовий файл з назвою Lab09.txt та вивести в нього наступну інформацію про співробітника: ПІБ, дата народження, посада, стаж, розмір базової зарплатні та розмір повної зарплатні (з урахуванням стажу та посади). Записи в текстовому файлі мають розташуватись в порядку зменшення повної зарплати (від найбільшої – до найменшої). Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 15

1) Створити клас з ім'ям `Person`. В даному класі створити поля в яких міститься ПІБ людини та дата її народження. Задати конструктор та деструктор класу.
2) В класі `Person` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури або зберігає передані дані про людину та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить відповідні дані на консоль.
3) За допомогою успадкування на основі класу `Person` створити клас-нащадок `Passenger` (пасажир), в якому додати поле з назвою станції відправлення (`Start`), станції прибуття (`Finish`) та відстані між ними (`Distance`).
4) В класі `Passenger` реалізувати метод, який визначає вартість квитка на проїзд за наступними правилами: при відстані подорожі від 100 до 500 км тариф становить 10 грошових одиниць/км, при відстані від 500 до 1000 – 7 одиниць/км, при відстані від 1000 до 1500 км – 5 одиниць/км.
5) Перевантажити метод `Input()` для класу `Passenger`. Метод має додатково запитувати/зберігати дані для полів `Start` та `Finish` і генерувати випадковим чином значення поля `Distance` за допомогою функції `rand()` з інтервалом значень 100–1500 км. Перевантажений метод `Output()` класу `Passenger` має додатково друкувати інформацію про маршрут пасажира та вартість квитка.
6) Реалізувати оператор виведення в потік `<<` даних про пасажира.
7) Створити і заповнити даними 5 об'єктів класу `Passenger`.
8) Створити текстовий файл з назвою Lab09.txt та вивести в нього всю наявну інформацію про пасажирів. Записи в текстовому файлі розташувати в порядку збільшення відстані подорожі. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 16

1) Створити клас `Point2D`, який описує точку на площині з декартовими координатами *x*, *y*. Поля, які містять координати, повинні мати специфікатор доступу `protected`. Задати конструктор та деструктор класу.
2) В класі `Point` реалізувати віртуальні методи для встановлення заданих координат точки та виведення цих координат на консоль. Реалізувати віртуальний метод `Init()`, який задає координати точки за допомогою функції `rand()` як випадкові числа в межах від 0 до 100.
3) За допомогою успадкування на основі класу `Point2D` утворити похідний клас `ColorPoint3D`, який додатково містить третю координату та поле, що задає колір точки (забезпечити можливість вибору не менше 5 різних кольорів).
4) В класі `ColorPoint3D` перевизначити методи для встановлення даних точки (координат і кольору) та виведення цієї інформації про точку.
5) Реалізувати оператор виведення в потік `<<` даних про точку.
6) Створити масив із 4 елементів типу `ColorPoint3D`, заповнивши вміст кожного об'єкту довільними даними. Реалізувати дружню функцію, яка визначає кількість різних кольорів точок, що присутні в масиві.
7) Створити дружню функцію, яка перевіряє, чи точки, що задані в масиві з п. 6, лежать в одній площині, чи ні. Продемонструвати результат роботи цієї функції.
8) Створити текстовий файл з назвою Lab09.txt та вивести в нього наступну інформацію про точки: тип точки, координати точок відповідного типу, їх колір. Порядок розташування записів в текстовому файлі має відповідати порядку кольорів точок в масиві, від найпершого кольору до останнього. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 17

1) Створити клас `Polynom5`, який описує поліном п'ятого порядку. Поля даного класу повинні містити коефіцієнти відповідного поліному, а також символьне позначення змінної (наприклад, `'x'`, або `'y'`). Задати конструктор та деструктор класу. В конструкторі за замовчуванням `Polynom5()` коефіцієнти полінома мають задаватись нульовими, а символьне позначення змінної має задаватись як `'y'`.
2) В класі `Polynom5` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури або зберігає передані дані про поліном (коефіцієнти та позначення змінної) та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить на консоль поліном у вигляді формули, на кшталт 8y^5 – y^4 +y^3 + 2y^2 – 6y + 3.
3) В класі реалізувати віртуальний метод, який обчислює похідну *n*-го

порядку ($n \geq 0$) від вказаного поліному і виводить її на екран у вигляді формули. Для збереження та виведення інформації про похідну скористатись функціоналом розробленого класу `Polynom5`.

4) За допомогою успадкування на основі класу `Polynom5` створити клас-нащадок `Polynom5Simple`, в якому коефіцієнт при змінній в степені 2 вважається нульовим. Перевантажити для класу `Polynom5Simple` метод для обрахунку похідної від полінома.
5) Реалізувати оператор виведення в потік `<<` даних про поліноми.
6) Створити 2 об'єкти класу `Polynom5` та 2 об'єкти класу `Polynom5Simple` і заповнити їх даними.
7) Створити текстовий файл з назвою Lab09.txt та вивести в нього у вигляді формул самі поліноми, пов'язані з об'єктами класу `Polynom5` та `Polynom5Simple` та похідні від цих поліномів. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 18

1) Створити клас з ім'ям `Tree`. В даному класі створити закриті поля, в яких міститься назва (виду) та вік дерева. Задати конструктор та деструктор класу.
2) В класі `Tree` реалізувати віртуальний метод `Input()`, який зчитує з клавіатури або зберігає передані дані про дерево та записує їх у поля класу, а також віртуальний метод `Output()`, який виводить на консоль інформацію про дерево.
3) За допомогою успадкування на основі класу `Tree` створити клас-нащадок `Fruit_Tree`, в якому додати динамічний масив `Productivity` значень типу `int`, що зберігає дані про щорічну врожайність фруктового дерева. Заповнити дані щодо врожайності з урахуванням раніше введеного віку дерева. Щорічна врожайність може змінюватись в межах від 50 до 300.
4) В класі `Fruit_Tree` реалізувати метод, який розраховує середню врожайність фруктового дерева. Реалізувати метод, який визначає та повідомляє про врожайність фруктового дерева: якщо середня врожайність < 100 виводиться характеристика низька, якщо вона між 100 та 200 – середня, при > 200 – висока.
5) Реалізувати оператор виведення в потік `<<` даних про дерева.
6) Створити 2 об'єкти класу `Fruit_Tree` і заповнити їх даними (варто обмежувати вік дерев 10 роками).
7) Створити текстовий файл з назвою Lab09.txt та вивести в нього всю наявну інформацію про дерева. Порядок розташування записів в текстовому файлі має бути від найбільш врожайних до найменш врожайних дерев. Відформатувати дані в зручному для читання вигляді. Вивести

аналогічним чином відформатовані дані на консоль.

Варіант 19

1) Створити клас з ім'ям `Site`. В даному класі створити захищені поля, в яких міститься URL адреса сайту (наприклад https://iht.knu.ua), рядок який містить короткий опис призначення сайту, рік створення, середню кількість відвідувачів за день. Задати конструктор та деструктор класу.
2) В класі `Site` реалізувати віртуальний метод, який визначає за доменною зоною в URL адресі країну, до якої відноситься сайт.
3) Використовуючи успадкування на основі класу `Site` створити клас-нащадок `SiteUA`, в якому додати ще одне поле зі значенням пінгу в мілісекундах. Щоб дізнатися значення пінгу для деякого сайту, в командному рядку наберіть ping та символьну адресу сайту, після чого натисніть Enter, наприклад: > ping google.com.
4) В класі `SiteUA` перевантажити функцію для визначення доменної зони (тут буде завжди .ua – Україна) таким чином, щоб вона видавала всю наявну інформацію про сайт.
5) Реалізувати оператор виведення в потік << даних про сайти.
6) Створити і заповнити даними об'єкти, що містять інформацію про 3 іноземні та 3 українські сайти. Використовувати по можливості реальні дані сайтів, кількість відвідувачів задавати як випадкову величину.
7) Створити текстовий файл з назвою Lab09.txt та вивести в нього інформацію про сайти: перші три українські сайти, відсортовані за величиною пінгу, та інші три сайти, відсортовані за алфавітом за назвою країни. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

Варіант 20

1) Створити клас з ім'ям `Chem_Reactor`. В даному класі створити поля, в яких міститься інформація про виробника хімічного реактора (символьний рядок), об'єм реактора (тип `float`), максимальна температура (тип `int`). Задати конструктор та деструктор класу.
2) В класі `Chem_Reactor` реалізувати віртуальну функцію, яка виводить інформацію про хімічний реактор. Функція має надати змогу вибрати формат виведення даних: об'єм реактора – в літрах (л) або кубічних метрах ($м^3$), максимальна температура – в градусах Цельсія (^oC), чи градусах Кельвіна (К).
3) Використовуючи успадкування на основі класу `Chem_Reactor` створити похідний клас `Vacuum_Chem_Reactor`, в якому додати ще два поля зі значенням мінімального та максимального тиску в реакторі.
4) Перевантажити у класі `Vacuum_Chem_Reactor` метод для виводу

інформації про реактор таким чином, щоб метод виводив і інформацію про робочі тиски. Надати можливість вибору формату виводу для тиску в барах чи в паскалях (Па).

5) Реалізувати оператор виведення в потік << даних про хімічний реактор.
6) Створити і заповнити даними 5 об'єктів класу `Vacuum_Chem_Reactor`.
7) Створити текстовий файл з назвою Lab09.txt та вивести в нього інформацію про наявні вакуумні хімічні реактори за алфавітним порядком назви виробника. Відформатувати дані в зручному для читання вигляді. Вивести аналогічним чином відформатовані дані на консоль.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. В чому полягає принцип успадкування? Для чого застосовується успадкування в мові C++? Наведіть приклади.
2. Для чого потрібен ключ успадкування? Яким чином визначається рівень доступу до елементів класу з урахуванням ключа успадкування?
3. Що таке просте успадкування? Множинне успадкування? Наведіть приклади.
4. Чи може специфікатор доступу до одного і того ж самого елемента відрізнятись в базовому та похідному класах? Наведіть приклади.
5. Чи може метод похідного класу викликатись через вказівник на базовий клас? Метод базового класу викликатись через вказівник на похідний клас? Відповідь пояснити.
6. Що таке віртуальна функція (метод)? Чим такі методи відрізняються від звичайних (не віртуальних) методів? Наведіть приклади.
7. Що таке абстрактні класи і для чого їх використовують? Наведіть приклади.
8. Для чого потрібні простори імен? Наведіть приклади.
9. Чим відрізняються вказівники на функцію і вказівники на методи класу?
10. Що таке дружні функції? Дружні класи? Поясніть для чого вони потрібні, наведіть приклади їх використання.
11. Поясніть принцип роботи технології перевантаження операцій. Які є обмеження щодо перевантаження операцій? Наведіть приклад використання перевантажених операторів.
12. Для чого в C++ використовуються нові правила приведення типів? Чому не завжди можна обійтись правилами приведення типів мови C?
13. Яким чином функціональність потоків C++ може бути розширена для роботи з даними довільного типу? Наведіть приклади.
14. В чому відмінність при роботі з текстовими та бінарними файлами при використанні файлових потоків C++? Наведіть приклади.
15. Які ви знаєте способи форматування даних при введенні/виведенні інформації за допомогою потоків C++? Наведіть приклади.

Лабораторна робота № 10.
Шаблони C++. Обробка винятків (виключень) C++. Ознайомлення з бібліотекою шаблонів STL

Мета роботи:

1) Навчитись створювати і застосовувати шаблони функцій і класів.
2) Ознайомитись з технологією обробки винятків (виключень) C++. Навчитись генерувати та обробляти винятки C++.
3) Ознайомитись з бібліотекою шаблонів STL. Навчитись працювати зі стандартними векторами, рядками, списками бібліотеки STL.

ШАБЛОНИ C++

Концепція шаблонів у мові C++

Функції і класи є характерними «будівельними блоками» для створення ефективних програм мовою C++. Проте в деяких випадках використання таких «блоків» може бути не дуже зручним через необхідність вказувати *конкретні* типи даних для параметрів функцій і методів, типи значень, що повертаються функціями/методами, а також типи для полів класів. Для того щоб «уникнути» такої деталізації щодо типів, у мові C++ існує технологія ***шаблонів***, яка дозволяє створювати класи та функції, задаючи для них типи як *параметри*. Завдяки цьому, шаблони C++ іноді називають ***параметризованими типами***.

На практиці досить часто трапляється ситуація, коли потрібно мати функції, які реалізують один і той же самий алгоритм обробки для даних різних типів. Для створення таких функцій, в принципі, можна використати технологію перевантаження функцій (див. лабораторну роботу № 3), написавши кілька однотипних однойменних функцій, кожна з яких працює лише з даними певного типу. Але в цьому випадку доведеться створити в програмі стільки «версій» функції, скільки різних типів даних необхідно обробляти. При цьому кожного разу єдиною відмінністю коду таких функцій буде лише тип даних, що використовується. Багаторазове переписування коду таких функцій, що відрізняються тільки типом даних, забирає час. Крім того, записаний код буде важко підтримувати та модифікувати, оскільки кожна з написаних функцій буде незалежною, незв'язаною з іншими аналогічними функціями. Будь-які модифікації в тілі таких функцій повинні реалізовуватись окремо для кожної функції. Це потребуватиме додаткового часу і збільшить ймовірність появи помилок.

Для елегантного та зручного розв'язання подібних задач в програмах на C++ існують шаблони, в яких використовується деякий *узагальнений тип* (або

типи). Шаблони являють собою звичайний фрагмент коду, в якому визначається функція або клас, де замість конкретних типів даних використовуються узагальнені типи. Таким чином, шаблон являє собою деяку «модель» або «заготовку» для функції/класу, записану для узагальненого (невизначеного) типу даних.

Якщо програмісту необхідно скористатись розробленим шаблоном, він вказує компілятору C++, що необхідно реалізувати відповідну функцію/клас з шаблону, використовуючи певний конкретний тип даних. На підставі цього компілятор C++ автоматично генерує код функції/класу з шаблону, заміняючи в шаблоні узагальнений тип на вказаний програмістом конкретний тип даних. Це дає можливість написати код шаблону лише один раз і потім використовувати його для роботи з довільним типом даних, що пришвидшує розробку програм, робить їх код більш уніфікованим. Крім того, спрощується підтримка та модернізація програм, що використовують шаблони, оскільки контролювати та за потреби змінювати слід лише код шаблону.

Шаблони функцій

Шаблонна функція (англ. template function) – це функція, яка реалізує певний алгоритм для довільного типу даних (як стандартних, так і введених програмістом). Шаблонна функція оголошується та визначається так само як і звичайна функція, але з додатковим початковим заголовком

```
template <список аргументів шаблону> ,
```

в якому зазначаються узагальнені типи, які використовуються в функції. За традицією ці узагальнені типи позначаються великими буквами (`T`, `U`, `X` тощо), перед якими вказується службове слово `class`[1] або `typename`. В тілі шаблонної функції узагальнені типи даних можна використовувати за тими ж правилами, що й відомі типи даних: створювати об'єкти узагальненого типу, викликати їх методи, передавати їх функціям/методам, реалізовувати різні операції мов C/C++ з цими об'єктами тощо.

Лістинг 10.1 демонструє створення та використання двох шаблонних функцій для додавання двох величин однакового та різного типів:

Лістинг 10.1:

```
1: #include <iostream>
2: #include <string>
3:
4: using namespace std;
```

[1] Записуючи параметри шаблонів, службове слово `class` слід інтерпретувати як позначення *деякого типу* (не обов'язково класу). Воно є аналогом `typename`.

```
 5:
 6: template <class T> // Заголовок шаблону.
 7: T Sum( T a, T b )
 8: {
 9:     return (a + b);
10: }
11:
12: // Перевантажений шаблон.
13: template <typename T, typename X>
14: T Sum( T a, X b )
15: {
16:     return (a + (T)(b));
17: }
18:
19: int main()
20: {
21:     // Використання шаблону для int.
22:     cout << "int:         4 + 8 = ";
23:     cout << Sum(4, 8) << endl;
24:     // Використання шаблону для double.
25:     cout << "double:      7.56 + 21.434 = ";
26:     cout << Sum(7.56, 21.434) << endl;
27:     // Використання шаблону для string.
28:     string s1("We use ");
29:     string s2("templates!");
30:     string s3 = Sum(s1, s2);
31:     cout << "string: " << s1 << "+ " << s2;
32:     cout << " = " << s3 << endl << endl;
33:
34:     // Використання шаблону для int/char.
35:     cout << "int/char:   17 + 'a' = ";
36:     cout << Sum(17, 'a') << endl;
37:     // Використання шаблону для double/int.
38:     cout << "double/int: 4.7 + 3 = ";
39:     cout << Sum(4.7, 3) << "\n\n\n" << endl;
40:
41:     return 0;
42: }
```

У лістингу 10.1 вводяться дві шаблонні функції, які до того ж виявляються перевантаженими: `T Sum( T a, T b )` і `T Sum( T a, X b )`. Перша з них, `T Sum( T a, T b )`, реалізує додавання двох величин однакового узагальненого типу `T` (рядки 6-10). Інша (рядки 13-17) – виконує додавання двох величин різних узагальнених типів `T` та `X`. Обидві функції

використовують операцію додавання `+`, а функція `T Sum( T a, X b )` ще використовує операцію приведення типу `(T)`. Це означає, що практична реалізація цих функцій буде можливою тільки для тих типів `T` та `X`, для яких дозволені відповідні операції.

В тілі функції `main()` демонструється використання шаблонних функцій для обрахунку суми величин різного типу. В рядку 23 виклик функції `Sum(4, 8)` інтерпретується компілятором як виклик `T Sum( T a, T b )`, де узагальнений тип `T` замінюється на конкретний тип `int`. Аналогічним чином виклик `Sum(7.56, 21.434)` в рядку 26 змушує компілятор реалізувати функцію `T Sum( T a, T b )`, в якій тип `T` перетворюється на тип `double`. Найбільш цікавим є наступний блок коду (рядки 28-30), де шаблон `Sum( T a, T b )` використовується для обрахунку «суми» двох текстових рядків, збережених як об'єкти класу `std::string`. Зазначимо, що в класі `std::string` реалізована перевантажена операція додавання `operator+()`, результатом виконання якої є новий об'єкт класу `std::string`, що включає в себе вміст обох операндів-об'єктів класу `std::string`. Тому фрагмент коду `(a + b)` в тілі шаблонної функції (рядок 9) компілюється успішно для параметрів функції, заданих у вигляді об'єктів класу `std::string`. Результатом виконання `Sum(s1, s2)` є текстовий рядок `"We use templates!"`, що зберігається в об'єкті класу `std::string`.

В рядках 35-39 використовується інша шаблонна функція `T Sum( T a, X b )`. Виклик `Sum(17, 'a')` інтерпретується компілятором як виклик, де тип `T` має бути типом `int`, а тип `X` – типом `char`. Виклик `Sum(4.7, 3)` змушує компілятор реалізувати функцію `T Sum( T a, X b )`, де тип `T` замінюється на `double`, а тип `X` – на `int`. Звернемо увагу, що при такому підборі типів, операція приведення типу `(T)` в рядку 16 є дозволеною і вже реалізованою в C++.

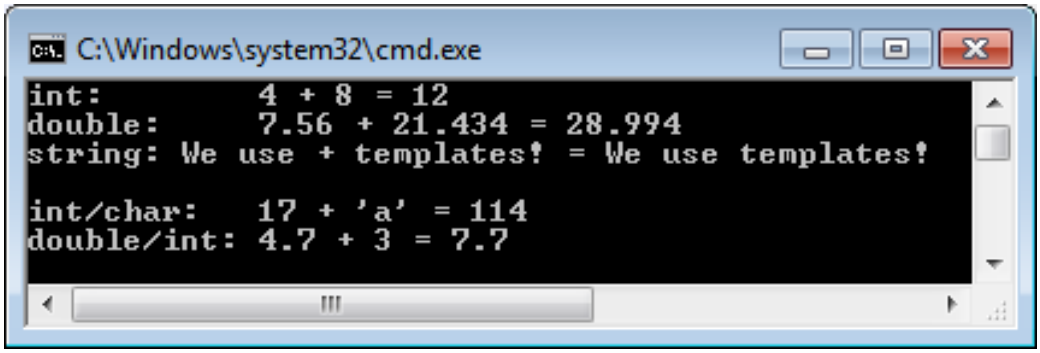


Рис. 38

Результати роботи програми з лістингу 10.1 показано на рис. 38.

Як видно з лістингу 10.1 шаблони функції дозволяють зекономити час, оскільки їх треба написати лише один раз, а потім можна використовувати з різними типами даних. При цьому немає необхідності, як при традиційному

перевантаженні функції, багато разів копіювати код, змінюючи лише тип даних. Крім того, шаблонні функції можуть працювати як зі стандартними типами даних (`char`, `int`, `double` тощо), так і з класами (наприклад, з класом `std::string`, який застосовується в лістингу 10.1). Але при роботі з класами (або іншими нестандартними типами) вбудовані оператори C/C++, такі як `+`, `-`, `<`, оператор виведення в потік `<<` тощо, мають бути перевантажені належним чином, щоб працювати з об'єктами нестандартних типів.

Шаблони класів

Шаблонний клас (англ. template class) – це клас C++, який реалізує збереження та роботу з даними довільного типу. Створення шаблонного класу аналогічне створенню шаблонної функції. При оголошенні шаблонного класу спочатку записується додатковий заголовок

```
template <список аргументів шаблону> ,
```

в якому зазначаються узагальнені типи даних, що використовуються в шаблоні. Далі безпосередньо йде тіло класу, в якому вводяться потрібні поля і методи, зокрема, поля і методи, що використовують узагальнені типи, пов'язані з шаблоном.

Для практичної реалізації коду шаблонного класу, необхідно явно вказати які саме типи слід підставляти у шаблон класу. Після того, як ця інформація зазначена, компілятор автоматично створює новий клас, побудований на основі вказаного шаблону. Щоб спростити подальшу роботу з таким класом, досить часто використовують перейменування типу за допомогою `typedef`. Введення та використання шаблонних класів на прикладі класу для роботи з комплексними числами демонструється в лістингу 10.2.

Лістинг 10.2:

```
 1: #include <math.h>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: template <typename T>
 7: class Complex
 8: {
 9:     T re, im;
10:
11: public:
12:     Complex(): re(0), im(0) {}
```

```
13:     Complex( T _re )
14:     {
15:         re = _re;
16:         im = 0;
17:     }
18:     Complex( T _re, T _im )
19:     {
20:         re = _re;
21:         im = _im;
22:     }
23:     Complex( const Complex& r )
24:     {
25:         re = r.re;
26:         im = r.im;
27:     }
28:     ~Complex(){}
29:
30:     // Доступ до даних.
31:     T Re() const { return re; }
32:     T Im() const { return im; }
33:
34:     // Зміна даних.
35:     void Re( T _re ) { re = _re; }
36:     void Im( T _im ) { im = _im; }
37:
38:     // Інші методи.
39:     double Module() const
40:     {
41:         return sqrt(re*re + im*im);
42:     }
43:     double Arg() const
44:     {
45:         return atan2(im, re);
46:     }
47:
48: public:
49:     // Перевантажені оператори.
50:     Complex operator+=( T a )
51:     {
52:         re += a;
53:         return *this;
54:     }
55:     Complex operator+=( const Complex& a )
56:     {
57:         re += a.re;
```

```
 58:           im += a.im;
 59:           return *this;
 60:       }
 61:       Complex operator-=( T a )
 62:       {
 63:           re -= a;
 64:           return *this;
 65:       }
 66:       Complex operator-=( const Complex& a )
 67:       {
 68:           re -= a.re;
 69:           im -= a.im;
 70:           return *this;
 71:       }
 72: };
 73:
 74: // Глобальні перевантажені оператори.
 75: template <typename T>
 76: Complex<T> operator+( const Complex<T>& a, const
 77: Complex<T>& b )
 78: {
 79:     return Complex<T>(
 80:         a.Re()+b.Re(), a.Im()+b.Im());
 81: }
 82:
 83: template <typename T>
 84: Complex<T> operator+( T a, const Complex<T>& b )
 85: {
 86:     return Complex<T>(a+b.Re(), b.Im());
 87: }
 88:
 89: template <typename T>
 90: Complex<T> operator+( const Complex<T>& a, T b )
 91: {
 92:     return Complex<T>(a.Re()+b, a.Im());
 93: }
 94:
 95: template <typename T>
 96: ostream& operator<<( ostream& os, Complex<T>& r )
 97: {
 98:     os << "Complex(" << r.Re() << "; ";
 99:     os << r.Im() << ")";
100:     return os;
101: }
102:
```

```
103:  typedef Complex<double> DComplex;
104:
105:  int main()
106:  {
107:      DComplex c1(2.5, -3.6);
108:      cout << c1 << endl;
109:
110:      DComplex c2(-0.4, 1.65);
111:      cout << c2 << endl;
112:
113:      cout << "c1 + c2 = ";
114:
115:      DComplex c3 = c1 + c2;
116:      cout << c3 << endl;
117:
118:      cout << "Module   of " << c3 << " is ";
119:      cout << c3.Module() << endl;
120:      cout << "Argument of " << c3 << " is ";
121:      cout << c3.Arg() << endl << endl << endl;
122:
123:      return 0;
124:  }
```

Алгоритм роботи програми з лістингу 10.2 принципово нічим не відрізняється від алгоритму роботи програми з лістингу 9.5. Суттєвою відмінністю між цими двома програмами є лише використання різних технологій мови C++. Програма з лістингу 9.5 використовує звичайний клас C++, а програма з лістингу 10.2 – шаблонний клас.

У рядках 6-72 оголошується шаблонний клас `Complex`. Його оголошення починається з обов'язкового рядка `template <typename T>`. В класі вводяться поля `re` та `im` узагальненого типу `T`, а також реалізуються конструктори, деструктор, перевантажені оператори та інші методи. При записі конструкторів поля `re` та `im` ініціалізуються значенням `0`, яке легко конвертується у значення будь-якого стандартного типу (`double`, `float`, `int`, `long`, `char` тощо).

При записі тіла методів всередині оголошення шаблонного класу додатково вказувати рядок `template <typename T>` непотрібно. Проте при визначені методів та функцій поза межами шаблонного класу обов'язково зазначати заголовок шаблону `template <typename T>`, а також вказувати *повне* ім'я шаблонного класу: `Complex<T>`. За цим принципом записаний код глобальних перевантажених операторів `operator+()`, `operator<<()`, що здатні працювати з шаблонним класом `Complex<T>` (рядки 75-101).

В рядку 103 вводиться нове ім'я `DComplex` для шаблонного класу

`Complex<T>`, побудованого на основі типу `double`. Далі, в тілі функції `main()`, клас `DComplex` використовується у той же спосіб, що й клас `CComplex` у лістингу 9.5. Результат роботи програми з лістингу 10.2 виявляється ідентичним результату роботи програми з лістингу 9.5 (див. рис. 36).

Винятки (виключення) C++

Винятки (виключення) в програмах на C/C++

Винятки (виключення) (англ. exception) – це повідомлення про появу певної особливої ситуації, що потребує негайної обробки. Сучасні процесори та ОС мають спеціальні засоби для генерування, відслідковування та обробки винятків, які можуть генеруватись як на апаратному рівні, так і за допомогою програмного коду.

Відмінністю винятку від звичайної помилки, що виникає в процесі роботи програми, є те, що виняток неможливо ігнорувати[1]. При появі винятка, нормальна робота програми відразу ж переривається, і процесор починає виконувати код для обробки відповідного винятку[2]. Тільки коли виняток оброблений, нормальна робота програми може бути відновлена.

В програмуванні на C/C++ розрізняють два види винятків. В мові C, зазвичай, використовують т. зв. ***структурні винятки (виключення)*** (structured exception). Саме ці винятки здатний генерувати та обробляти процесор. Також ці винятки переважно використовуються компонентами ОС, драйверами пристроїв, програмами з прямим доступом до апаратних ресурсів комп'ютера тощо. Ці винятки є *низькорівневими*. На основі низькорівневих структурних винятків реалізуються *високорівневі* ***винятки (виключення) C++***, які будуть розглядатись далі.

Винятки C++ часто використовуються як повідомлення про появу серйозних помилок, що виникають під час роботи програми. Хоча винятки можна

---

[1] Насправді, деякі виключення, але не всі, можна заблокувати, тим самим заборонивши їх обробку. Проте існує велика кількість виключень, як правило, пов'язаних з відмовою або невірною роботою апаратних компонент комп'ютера, які неможливо заблокувати.

[2] При виникненні виключення, коли переривається нормальна робота програми, розпочинається процес пошуку ***обробника*** відповідного виключення. Обробники виключень утворюють певну ієрархію: *обробник програми – обробник стандартної бібліотеки C/C++ – обробник ОС*. Обробник програми створюється програмістом підчас написання тексту програми. Він є першим обробником, який має змогу обробити виключення, що виникає. Якщо такий обробник відсутній, управління передається більш зовнішньому обробнику – обробнику, реалізованому в стандартній бібліотеці C/C++. Якщо і цей обробник не в змозі обробити виключення, або якщо підчас його роботи виникають нові виключення, управління може бути передано обробнику ОС, який, зазвичай, аварійно завершує роботу програми і виводить характерне для ОС повідомлення про збій програми.

використовувати і за іншим призначенням, слід пам'ятати, що їх генерація та обробка потребують виконання досить складного коду. Тому надмірне використання винятків в програмі може помітно знизити її швидкодію. Разом з тим, використання винятків C++ в програмах дозволяє отримати і цілий ряд переваг:

- На відміну від звичайних помилок, винятки не можна ігнорувати. Якщо деякий виняток C++ в програмі не обробляється, його обробка буде обов'язково передана більш зовнішньому обробнику винятків з бібліотеки C++ або ОС.
- Винятки є невід'ємною частиною мови C++. Вони підтримуються всіма сучасними процесорами та ОС, що спрощує налагодження програм, де застосовуються винятки.
- Програми, написані з використанням винятків, є більш структурованими, уніфікованими. В них явним чином виділяються ті фрагменти тексту програми, де можлива генерація винятків, і ті фрагменти, де відбувається обробка винятків, завдяки чому такі програми легше модернізувати та супроводжувати.
- Генерування винятку в тілі функції/методу є особливою формою завершення її роботи. Іноді використання винятків – це єдиний зручний спосіб повідомити про появу помилки всередині функції/методу. Наприклад, це є типовим підходом при появі помилок всередині конструкторів класу, які ніколи не повертають значень, або перевантажених операторів, які повертають лише значення певного типу.

Генерація та обробка винятків в C++

Обробка виключень в мові C++ реалізована за допомогою трьох ключових слів: `throw`, `try` і `catch`.

Для того, щоб сигналізувати про виникнення винятку використовується ***оператор*** `throw`. Процес утворення винятку називається ***генерацією*** або ***викиданням винятку (виключення)***. Для генерації винятку C++ в програмі записується ключове слово `throw`, після якого вказується певне значення (об'єкт) довільного типу даних, яке буде використано, щоб повідомити про появу конкретного винятку. Це може бути числовий код помилки або текстовий опис проблеми, яка виникла, або навіть об'єкт певного структурованого типу. Наприклад, наступний фрагмент коду викидає текстовий рядок, що містить інформацію про неприпустиме значення параметра `a`:

```
double Sqrt( double a )
{
    if( a < 0.0 ) throw "a is a negative number";
    else return sqrt(a);
```

```
}
```

Оператор `throw` може використовуватись й без операнда. В цьому випадку той виняток, який обробляється в даний момент (наприклад, в блоці `catch` – див. далі), генерується знову.

Ключове слово `try` виділяє блок операторів, який використовується для *відслідковування* винятків. Такий блок операторів, записаний у «дужках» `try`

```
try
{
    // Деякі оператори...
}
```

називається ***захищеним блоком***. Часто його також називають ***блоком try***. Цей блок `try{/*...*/}` виконує роль «спостерігача», який відстежує винятки, що можуть бути згенеровані одним з операторів в тілі цього блоку. Цей блок також буде реагувати на будь-які винятки, згенеровані в довільній функції або методі, що викликаються всередині блоку. При цьому блок `try` *не визначає*, що саме слід робити з винятком, який в ньому може виникнути. Для обробки винятків застосовуються блоки `catch`.

Фактична обробка винятків відбувається у блоці `catch`. Ключове слово `catch` відмічає блок коду, в якому обробляється виняток, пов'язаний з певним типом даних. Цей тип даних вказується в круглих дужках після ключового слова `catch`. Наприклад, наступний блок, обробляє виняток, пов'язаний з типом `int`, або, як часто говорять, обробляє виняток типу `int`:

```
catch( int a )
{
    cerr << "An int exception with value";
    cerr << a << endl;
}
```

Блоки `try` та `catch` використовуються разом. Блок `try` відстежує винятки, які виникли всередині нього, а блок `catch`, який записується відразу після блоку `try`, використовується для обробки винятків. Блок `try` повинен обов'язково супроводжуватися одним або декількома блоками `catch`, які розташовані після блоку `try`. При цьому сам блок `catch` може бути і порожнім, головне, щоб він був присутнім в тексті програми.

Після блоку `try` може бути розташовано декілька блоків `catch`, що обробляють винятки, пов'язані з різними типами даних (це досить загальна практика). Після генерації винятку його обробка відбувається в першому ж блоці `catch`, який пов'язаний з відповідним типом даних винятку. Отже, порядок запису блоків `catch` має важливе значення. Крім того, слід пам'ятати про

правила автоматичного приведення типу в C/C++, неврахування яких також може призвести до невірного порядку обробки винятків. Приклад відповідної логічної помилки наведено в лістингу 10.3.

Лістинг 10.3:

```
 1: #include <iostream>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     try
 8:     {
 9:         throw "Exception";
10:     }
11:     catch( void* )
12:     {
13:         cout << "In catch( void* )";
14:         cout << endl;
15:     }
16:     catch( char* )
17:     {
18:         cout << "In catch( char* )";
19:         cout << endl;
20:     }
21:
22:     return 0;
23: }
```

В тілі функції `main()` в рядку 9 генерується виняток, пов'язаний з текстовим рядком `"Exception"`, – тобто типом `char*`. Цей виняток має оброблятись в блоці `catch( char* )` (рядки 16-20). Проте тип `char*` *автоматично може бути приведеним* до типу `void*`, а блок `catch( void* )` іде *першим* після блоку `try`. Отже, хоча в блоці `try` генерується виняток, пов'язаний з `char*`, оброблятись від буде в блоці `catch( void* )`, а не в блоці `catch( char* )`. Тобто програма з лістингу 10.3 буде виводити на консоль рядок In catch(void*).

Ще однією логічною помилкою при використанні блоків `try-catch` є запис першого блоку `catch` у формі `catch(...){/*...*/}`. Блок `catch(...)` дозволяє обробляти винятки *будь-якого типу*, тому, якщо такий блок застосовується, він обов'язково повинен записуватись останнім. Інакше будь-які винятки будуть оброблятись не спеціалізованим блоком `catch`, налаштованим на певний тип даних, а єдиним універсальним блоком

`catch(...)`.

Виняток, який потрапив у блок `catch`, вважається *обробленим*. Після завершення відповідного блоку `catch` робота основної програми продовжується з оператора, що розташований одразу після блоку `catch`.

Розглянемо ще один приклад використання оператора `throw`, блоків `try` та `catch`:

Лістинг 10.4:

```
 1: #include <math.h>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:     double a;
 9:
10:     cout << "Enter a number: ";
11:     cin >> a;
12:
13:     try
14:     {
15:         if( a < 0.0 )    throw "Negative number";
16:         if( a == 0.0 )   throw 0.0f;
17:         if( a > 1.0E6 ) throw a;
18:         cout << "The square root of " << a;
19:         cout << " is " << sqrt(a) << endl;
20:     }
21:     catch( char* e )
22:     {
23:         cout << "Error: " << e << endl;
24:     }
25:     catch( double )
26:     {
27:         cout << a << " is too large" << endl;
28:     }
29:     catch( float )
30:     {
31:         cout << a << " is zero";
32:     }
33:     catch(...)
34:     {
35:         cout << "Unknown exception" << endl;
36:     }
```

```
37:
38:       cout << endl << endl;
39:
40:       return 0;
41:  }
```

В рядках 10-11 користувачу пропонують ввести дійсне число і зчитують введене число з консолі. В наступному блоці `try` (рядки 13-20) перевіряють введене число на «коректність». Якщо число від'ємне, викидається виключення у вигляді текстового рядка `"Negative number"`. Якщо введене число – це нуль, генерується виключення, пов'язане з типом `float`. Якщо ж число завелике (більше за `1.0E6`) – викидається виняток у вигляді самого цього числа (типу `double`). Таким чином, залежно від того, яке число введе користувач, можуть генеруватись винятки різного типу, які будуть оброблятись різними блоками `catch`, серед яких останнім блоком є блок `catch(...)`.

Робота з ієрархією винятків (виключень)

Як вже зазначалось раніше, винятки (виключення) часто застосовують, щоб обробляти помилки. При цьому іноді буває зручно з кожною помилкою пов'язати деякий клас, наприклад, клас `Error`, в якому створити поля та методи, корисні для отримання інформації про помилку, для обробки помилки тощо. Далі на основі класу `Error` можна реалізувати декілька похідних класів. Кожен з них буде мати принаймні таку ж саму функціональність, як і базовий клас `Error`, проте буде пов'язаним з певною конкретною помилкою, або групою помилок. За необхідності продовжуючи такий процес утворення класів-нащадків можна отримати *ієрархію* класів, виведених з єдиного базового класу `Error`. Кожен елемент такої ієрархії класів буде описувати певну помилку або тип помилки. Якщо тепер при роботі з винятками застосовувати клас `Error` або похідні від нього класи, по суті, отримаємо ***ієрархію винятків***, що повністю відображає ієрархію класів, породжених від класу `Error`[1].

У лістингу 10.5 наведено приклад створення та використання ієрархії винятків:

Лістинг 10.5:

```
1:  #include <iostream>
2:
3:  using namespace std;
4:
```

[1] З кожним винятком C++ пов'язується певний тип даних. В нашому прикладі – це клас `Error` або класи, похідні від `Error`.

```
 5: class Error
 6: {
 7: public:
 8:     virtual void Why()
 9:     {
10:         cout << "Error" << endl;
11:     }
12: };
13:
14: class FileError    : public Error
15: {
16: public:
17:     void Why()
18:     {
19:         cout << "File Error" << endl;
20:     }
21: };
22:
23: class FileReadError: public FileError
24: {
25: public:
26:     void Why()
27:     {
28:         cout << "File Read Error" << endl;
29:     }
30: };
31:
32: class MemoryError  : public Error
33: {
34: public:
35:     void Why()
36:     {
37:         cout << "Memory Error" << endl;
38:     }
39: };
40:
41: int main()
42: {
43:     try
44:     {
45:         FileReadError e;
46:         throw &e;
47:     }
48:     catch( FileReadError* p )
49:     {
```

```
50:             p->Why();
51:         }
52:         catch( FileError* p )
53:         {
54:             p->Why();
55:         }
56:         catch( MemoryError* p )
57:         {
58:             p->Why();
59:         }
60:         catch( Error* p )
61:         {
62:             p->Why();
63:         }
64:         catch( ... )
65:         {
66:             cout << "Unknown exception\n";
67:         }
68:
69:         cout << endl << endl;
70:
71:         return 0;
72:  }
```

У рядках 5-12 вводиться базовий клас `Error`, віртуальний метод `Why()` якого виводить текстове повідомлення про помилку (текстовий рядок `"Error"`). Від класу `Error` породжуються класи `FileError` (рядки 14-21) та `MemoryError` (рядки 32-39). Від класу `FileError` утворюється клас-нащадок другого рівня `FileReadError` (рядки 23-21). В усіх класах, похідних від `Error`, метод `Why()` перевантажується таким чином, щоб він виводив на консоль ім'я відповідного класу.

В тілі функції `main()` в блоці `try` (рядки 43-47) створюється об'єкт `e` класу `FileReadError`, який далі викидається за допомогою оператора `throw &e;`.

Після генерації винятку він обробляється одним із блоків `catch`, які записані в рядках 48-67. Оскільки в програмі з лістингу 10.5 генерується виняток, пов'язаний з типом `FileReadError*`, буде виконуватись тіло оператора `catch( FileReadError* p )`.

Як видно з наведеного прикладу, використовуючи ієрархію класів, пов'язаних між собою через успадкування, можна проводити вибіркову обробку різних помилок за допомогою різних блоків `catch`. Така поведінка може бути корисною при розробці бібліотек класів, для уніфікації системи обробки помилок в програмі тощо. Разом з тим, слід підкреслити, що відповідні

технічні прийоми, що ґрунтуються на використанні ієрархії класів, можуть використовуватись при роботі з винятками, які пов'язані з будь-якими подіями, не обов'язково з тими, які пов'язані з появою помилок.

ОЗНАЙОМЛЕННЯ З БІБЛІОТЕКОЮ STL

Принципи створення та можливості бібліотеки STL

Одним із недоліків «традиційного програмування» мовами C/C++ є необхідність багаторазового написання однотипного коду для збереження та обробки різних типів даних. Наприклад, для використання в програмі «традиційних» динамічних масивів, необхідно переписувати код, що реалізує роботу масиву, для кожного нового типу даних. Звичайно ж, цю проблему дозволяють усувати шаблони – достатньо лише один раз написати шаблон для динамічного масиву, а далі його можна легко адаптувати до будь-яких потреб (будь-яких типів). Однак, розробити ефективний шаблон досить непросто. І якщо для такого контейнера даних як динамічний масив це може зробити навіть програміст-початківець, розробити якісні шаблони для більш складних контейнерів (списків, стеку, хеш-таблиці тощо) може бути доволі складно. До того ж розроблені різними програмістами шаблони можуть суттєво відрізнятись за функціональністю. Наприклад, у деяких шаблонах може бути передбачена можливість сортування, вставки, зміни порядку розташування даних, а в інших шаблонах – ні.

Щоб полегшити роботу програміста і в той же час виконати певну уніфікацію алгоритмів збереження та обробки даних довільного типу, в кінці XX ст. була розроблена ***стандартна бібліотека шаблонів STL*** (standard template library).

Бібліотека STL – це частина стандартної бібліотеки C++, яка являє собою набір ***шаблонів класів-контейнерів*** даних, ***алгоритмів*** для роботи з даними та «вказівників» спеціального виду – ***ітераторів***. Код бібліотеки STL написаний професійними програмістами, багаторазово перевірений та ефективно реалізований. В бібліотеці STL реалізовані всі основні контейнери даних, які найчастіше зустрічаються на практиці (всі вони знаходяться в просторі імен `std`):

Контейнер:	Опис:
`array`	Є аналогом контейнера `vector`, проте має обмеження пов'язані із зміною кількості елементів.
`deque`	Черга з двостороннім доступом.
`list`	Двозв'язний список.
`map`	Карта або словник – контейнер з унікальних пар

Контейнер:	Опис:
	елементів «даних – ключ», розташованих у певному порядку.
`multimap`	Мультикарта або мультисловник – контейнер з пар елементів «даних – ключ», які можуть дублюватися і розташовані в певному порядку.
`multiset`	Мультимножина – це контейнер елементів-ключів, що можуть дублюватися, які розташовані в певному порядку.
`queue`	Черга – структура даних, що працює за принципом «перший зайшов – перший вийшов» (FIFO).
`set`	Множина – контейнер унікальних елементів-ключів, розташованих у певному порядку.
`stack`	Стек – структура даних, що працює за принципом «останній зайшов – першим вийшов» (LIFO).
`vector`	Аналог лінійного динамічного масиву, в якому всі елементи зберігаються послідовно в одному блоці динамічної пам’яті.

Для доступу до елементів даних, що зберігаються в контейнерах, як правило, використовуються не звичайні вказівники або операція індексації масиву `[]`, а спеціальні «вказівники» – *ітератори*. ***Ітератор*** являє собою спеціальний шаблонний клас, пов’язаний з шаблоном-контейнером. Ітератор дозволяє отримати безпечний доступ до даних, що зберігаються в контейнері, перейти до наступного або попереднього елементу даних тощо. Ітератори реалізуються різним чином для різних класів-контейнерів, проте забезпечують єдиний уніфікований інтерфейс для роботи з елементами даних контейнерів. Завдяки такій уніфікації алгоритми роботи з даними, записані з використанням ітераторів, виглядають ідентично (або дуже схоже) для будь-яких шаблонів-контейнерів.

Залежно від виду контейнера, зазвичай, виділяють кілька можливих для нього видів ітераторів:

- ***Вхідні ітератори*** забезпечують доступ до даних тільки в режимі для читання.
- ***Вихідні ітератори*** забезпечують доступ до даних тільки в режимі для запису.
- ***Однонаправлені ітератори*** дозволяють зчитувати та записувати дані при проходженні вмісту контейнера лише в одному напрямку (однонаправлений доступ).
- ***Двонаправлені ітератори*** дозволяють зчитувати та записувати дані при проходженні вмісту контейнера в обох напрямках, як в «прямому», так і в «зворотному» напрямку.
- ***Ітератори довільного доступу*** дозволяють зчитувати та записувати дані

при зверненні до будь-якого елементу контейнера.

Не кожен контейнер може реалізовувати всі зазначені види ітераторів. Наприклад, ітератор довільного доступу є дозволеним для контейнера `vector`, проте недоступний для списку. Двонаправлений ітератор є типовим для подвійної черги (`deque`), проте недоступний для контейнера `stack`.

Третім ключовим компонентом бібліотеки STL є ***алгоритми STL*** – шаблонні функції, які за допомогою ітераторів реалізують велику кількість (кілька десятків) алгоритмів обробки даних: сортування даних, їх перемішування, розташування даних у зворотному порядку, пошук даних, пошук даних за виконання певної умови, виконання певної операції для кожного елементу даних тощо. Завдяки використанню ітераторів, всі алгоритми STL виявляються універсальними, не залежними від виду контейнера даних. Відмінність коду алгоритма при роботі з різними контейнерами полягає лише у використанні різних ітераторів, кожен з яких пов'язаний з відповідним контейнером.

Використовуючи контейнерні класи, ітератори та алгоритми STL, можна досить швидко писати програми на C++, в яких застосовуються ефективні способи збереження та обробки даних довільного типу. Це робить бібліотеку STL важливим інструментом для швидкої розробки професійних програм на C++.

Бібліотека STL має значний об'єм. Детально вивчити всі її можливості можна лише в рамках окремого спеціалізованого курсу по C++ та/або за допомогою окремої книги, присвяченої саме STL. Враховуючи це, далі буде стисло розглянуто використання лише двох найбільш часто вживаних контейнерів STL: `vector` та `list`.

Контейнер std::vector

Шаблон `std::vector` – це контейнер, який містить набір лінійно впорядкованих елементів однакового типу та забезпечує швидкий доступ до довільного елементу даних. Цей клас оголошений у файлі заголовку `<vector>` і є більш зручною та безпечною версією розглянутих раніше динамічних масивів. На відміну від звичайних масивів, в яких не дозволяється зміна довжини масиву після того, як пам'ять для масиву була виділена, клас-контейнер `vector` може вільно керувати виділеною для нього динамічною пам'яттю. Завдяки цьому, використовуючи `vector`, можна створювати динамічні масиви змінної довжини без явного використання в програмі операторів `new[]` та `delete[]`.

Розглянемо різні способи створення об'єктів-векторів з цілих чисел:

```
#include <vector>
vector <int> v1;
vector <int> v2(3);
vector <int> v3(5, 2);
```

```
vector <int> v4 = { 10, 8, 6, 4, 2, 1 };
```

В першому випадку буде створено порожній вектор `v1`, який містить дані цілого типу `int`. В другому випадку буде створено вектор `v2`, який містить 3 елементи. Вектор буде заповнений значеннями за замовчуванням, які для даних типу `int` є числами `0`. В третьому прикладі буде створено вектор `v3`, який містить 5 елементів, зі значенням `2` кожний. В останньому випадку елементи створеного об'єкта `v4` задаються явним чином.

Зазначимо, що при використанні шаблона-контейнера `vector` немає необхідності явно вказувати довжину масиву. Більше того, цей клас дозволяє присвоювати нові значення вже існуючому об'єкту-вектору не поелементно, а *всі значення разом*, через перелік значень. В цьому випадку масив-вектор буде самостійно змінювати свою довжину так, щоб вона відповідала кількості заданих елементів:

```
v3 = { 0, 2, 4, 5, 7 };   // довжина v3 тепер 5
v3 = { 11, 9, 5 };        // довжина v3 тепер 3
```

Коли об'єкт-вектор знищується (наприклад, як локальний об'єкт при завершенні функції), він автоматично звільняє виділену для нього динамічну пам'ять. Таким чином, при роботі з класом `vector` немає необхідності в «ручному» контролі за динамічною пам'яттю об'єкта – відповідні операції звільнення пам'яті проводяться автоматично. Завдяки цьому використання `std::vector` є більш безпечним, ніж виділення/звільнення динамічної пам'яті через безпосередні виклики операторів `new[]` та `delete[]`.

На відміну від стандартних динамічних масивів, клас `std::vector` зберігає інформацію про власну довжину. Для отримання інформації про довжини масиву треба викликати метод `size()`:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> a = { 12, 10, 8, 6, 4, 2, 1 };
    std::cout << "The length:" << a.size() << endl;
    return 0;
}
```

Для зміни довжини для об'єкта типу `vector` можна використати метод `resize()`:

```
vector<int> a ={ 0, 1, 2 };
```

```
a.resize(7); // Змінюємо довжину a на 7.
```

При збільшенні довжини вектора, існуючі в ньому значення елементів зберігаються, а нові (додані) елементи ініціалізуються значенням за замовчуванням для відповідного типу даних (наприклад, `0` для типу `int`).

Ініціалізація елементів вектора також може відбуватися за допомогою метода `assign(count, Type& value)`. Цей метод вставляє у вектор набір заданих елементів або заповнює його певною кількістю копій одного і того ж самого елемента. Якщо вектор до виклику `assign()` містив якісь дані, вони будуть видалені, і лише потім у порожній вектор будуть вставлені нові дані, передані як параметри `assign()`.

Функція `push_back(Type& value)` додає до вектора новий елемент, який буде розташовуватись в кінці масиву (стане його останнім елементом):

```
vector<int> v1, v2;
v1.assign({ 5, 6, 7 });
v1.assign(7, 4);// Вже нові значення для вектора v1.
v2.assign(2, 0);// Заповнити вектор двома значеннями 0.
v2.push_back(1);// Додати значення 1 в кінець вектора.
v2.push_back(2);// Додати значення 2 в кінець вектора.
```

Метод `begin()` повертає значення ітератора типу `vector<T>::iterator`, який вказує на перший елемент типу `T` даного вектора. Метод `end()` повертає ітератор, який вказує на елемент, що стоїть в кінці вектора. Наступний фрагмент коду показує як вивести на екран значення першого елемента вектора – для цього слід лише розіменувати відповідний ітератор, що повертає `begin()`:

Лістинг 10.6:

```
 1: #include <iostream>
 2: #include <vector>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:     vector<int> v1;
 9:     vector<int>::iterator Iter;
10:
11:     v1.push_back(1);
12:     v1.push_back(2);
13:
14:     Iter = v1.begin();
```

```
15:       cout << *Iter;
16:
17:       return 0;
18:  }
```

В рядку 8 створюється об'єкт `v1`, який є вектором цілих чисел, а в рядку 9 створюється об'єкт ітератора `Iter` для контейнера `vector<int>`.

В рядках 11, 12 в контейнер записуються значення 1 та 2.

Використовуючи значення `v1.begin()` в рядку 14 відбувається ініціалізація ітератора `Iter`. Далі на консоль виводиться вміст елемента вектора, пов'язаного з відповідним ітератором. В результаті виконання `cout << *Iter;` на консоль виводиться значення першого елемента, на який вказує ітератор (тобто значення 1). Для безпосереднього доступу до значення елемента використовується перевантажена операція розіменування ітератора `*Iter`.

За допомогою ітераторів можна додавати елементи вектора не лише в кінці списку, а і в довільну позицію. Для цього застосовується метод `insert(iterator position, Type& value)`. Наприклад, фрагмент коду

```
v1.insert( v1.begin( )+1, 40 );
```

вставить другим елементом вектора число `40`, а фрагмент коду

```
v1.insert( v1.begin( )+2, 4, 50 );
```

вставить у вектор чотири числа `50`, починаючи з третього елемента, який має індекс 2.

Для видалення елементів вектора існують методи `pop_back()`, який видаляє останній елемент; `clear()`, який повністю видаляє всі елементи; та `erase()`, який видаляє елементи із вказаного інтервалу індексів. Так, оператор

```
v1.erase( v1.begin() );
```

видалить перший елемент вектора, а фрагмент коду

```
v1.erase( v1.begin() + 1, v1.begin() + 4 );
```

видалить елементи з другого по четвертий включно. Тут другий аргумент методу вказує не на останній видалений елемент, а на позицію одразу після нього (тобто на п'ятий елемент вектора, що має індекс `4`).

Наступний фрагмент коду показує приклад створення, заповнення та

виведення на консоль вмісту всіх елементів вектора:

Лістинг 10.7:

```
 1: #include <iostream>
 2: #include <vector>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:     vector<int> v1;
 9:     vector<int>::iterator It;
10:
11:     v1.push_back(10);
12:     v1.push_back(20);
13:     v1.push_back(40);
14:     v1.pop_back();
15:     v1.push_back(30);
16:
17:     // Виведення всіх елементів вектора.
18:     for( It = v1.begin(); It != v1.end(); It++ )
19:     {
20:         cout << " " << *It;
21:     }
22:
23:     return 0;
24: }
```

В рядку 8 лістингу 10.7 створюється об'єкт `v1`, який є вектором цілих чисел, а в рядку 9 створюється об'єкт-ітератор `It` для контейнера `vector<int>`.

В рядках 11-13 в контейнер послідовно додаються значення 10, 20 та 40. В рядку 14 з вектора видаляється останній елемент зі значенням 40, а на його місце записується елемент зі значенням 30 (рядок 15).

Використовуючи ітератор `It`, в рядках 18-21 відбувається перебирання всіх елементів вектора `v1` в циклі `for`. Початкове значення ітератора встановлюється як ітератор `v1.begin()`, що вказує на перший елемент вектора. Цикл має виконуватись поки не буде досягнуто т. зв. ***термінальний елемент***, який розташований відразу після останнього елемента вектора і на який вказує ітератор `v1.end()`. В кінці кожної ітерації циклу виконується перевантажений оператор `It++`, який переводить ітератор на наступний елемент вектора. Виконання операції `*It` в тілі циклу дозволяє одержати значення того елементу, на який в даний момент вказує ітератор `It`. В результаті роботи програми

на консоль виводяться значення 10, 20, 30.

Контейнер std::list

Клас `std::list` – це шаблон двозв'язного списку, що забезпечує послідовний двонаправлений доступ до елементів даних і дозволяє додавати або вилучати ці елементи в довільному місті списку. Клас `std::list` забезпечує більшу ефективність роботи при додаванні або видаленні елементів, ніж шаблон `vector`, хоча й програє по швидкодії вектору, коли слід отримати доступ до довільного елемента в контейнері.

Список певного розміру чи з певними елементами на основі `std::list` можна створити, користуючись тими ж принципами, що й для шаблону вектора:

```
// Створюється порожній список list0 цілих чисел.
list<int> list0;
// Створюється список list1 з трьома елементами
// (за замовчуванням мають значення 0).
list<int> list1(3);
// Створюється список list2 з 4 елементами
// зі значеннями 1.
list<int> list2(4, 1);
```

Багато методів класу `std::list`, зокрема, `assign()`, `begin()`, `end()`, `clear()`, `erase()`, `insert()`, `pop_back()`, `push_back()`, `size()`, працюють аналогічно до розглянутих вище методів класу `std::vector`, тому детально зупинятися на них не будемо. З оригінальних методів шаблону `std::list` відмітимо `push_front()` та `pop_front()`, які, відповідно, додають та видаляють елемент на початку списку; `remove()`, який видаляє зі списку всі елементи, які співпадають із вказаним в аргументі значенням, та `reverse()`, що змінює порядок елементів у списку на зворотній.

Наступний код показує приклад роботи з шаблоном списку:

Лістинг 10.8:

```
1: #include <iostream>
2: #include <list>
3:
4: using namespace std;
5:
6: int main()
7: {
```

```
 8:      list<int> list;
 9:      list<int>::iterator It;
10:
11:      list.push_back(10);
12:      list.push_back(20);
13:      list.push_back(30);
14:      list.push_front(5);
15:
16:      It = list.begin();
17:      cout << "The first element is ";
18:      cout << *It << endl;
19:      cout << "The size of the list is ";
20:      cout << list.size() << endl;
21:
22:      list.erase(list.begin());
23:
24:      It = list.begin();
25:      cout << "The first element now is ";
26:      cout << *It << endl;
27:
28:      list.clear();
29:      cout << "The size of list now is ";
30:      cout << list.size() << endl;
31:
32:      return 0;
33:  }
```

В лістингу 10.8 спочатку створюється об'єкт `list` списку з цілих чисел (рядок 8). У рядку 9 вводиться об'єкт-ітератор `It` типу `list<int>::iterator`.

В рядках 11-14 список заповнюється значеннями 10, 20, 30, які послідовно додаються за допомогою `push_back()` в кінець списку. Значення 5 при виклику `push_front()` додається на початок списку. Отже, вміст списку – елементи 5, 10, 20, 30.

За допомогою ітератора `It`, що задається як `list.begin()`, виводиться вміст першого елементу списку `*It` (рядки 16-18). Далі виводиться розмір списку, отриманий за допомогою `list.size()`.

В рядку 22 здійснюється видалення першого елементу списку. Потім вміст першого елементу списку виводиться знову (рядки 24-26).

При виклику `list.clear()` в рядку 28 всі елементи списку видаляються, отже, метод `list.size()` в рядку 30 повертає значення 0.

Сортування даних контейнера за допомогою STL

Розглянемо приклад використання алгоритму сортування даних бібліотеки STL.

Для сортування даних, що зберігаються в класах-контейнерах використовується функція `sort()`, опис якої знаходиться у файлі заголовку `algorithm`. Вона впорядковує елементи контейнера у зростаючому порядку (за замовчуванням), або згідно з критерієм, заданим спеціальним бінарним предикатом. Синтаксис цієї функції наступний:

```
void sort(Iterator first, Iterator last);
void sort(Iterator first, Iterator last, Compare pred);
```

Її аргументами виступають два ітератори довільного доступу, які вказують на положення першого `first` елемента в діапазоні сортування даних та елемента `last`, який стоїть одразу після останнього. Останнім, необов'язковим, аргументом функції є бінарний предикат – певна функція, що задає критерій порівняння, який буде враховуватися при упорядковуванні елементів контейнера. Функція-предикат приймає два аргументи (того ж типу, що зберігаються в контейнері) і повертає логічний результат `true`, якщо ці два аргументи розташовані в «правильному» порядку (тобто другий із них є «більшим» за перший), та `false` в іншому випадку. Власне, всередині тіла такої функції і потрібно встановити критерії порівняння двох довільних об'єктів потрібного типу.

Функція `sort()` не є специфічним методом якогось із контейнерів, але може бути застосована до будь-якого із них, якщо в якості її параметрів вказати відповідні ітератори.

Наступний код спочатку виведе на екран елементи контейнеру типу `vector`, відсортовані за зростанням *значення* елементу, а потім – елементи, відсортовані за зростанням ***модуля*** елементу:

Лістинг 10.9:

```
 1: #include <iostream>
 2: #include <vector>
 3: #include <algorithm>
 4:
 5: using namespace std;
 6:
 7: bool UDgreater( int elem1, int elem2 )
 8: {
 9:     if( abs(elem1) < abs(elem2) )
10:         return true;
11:     else return false;
```

```
12:  }
13:
14:  int main()
15:  {
16:       vector<int> v1;
17:       vector<int>::iterator It;
18:       int i;
19:
20:       for( i = 0 ; i <= 5 ; i++ )
21:       {
22:            v1.push_back(2*i*i - 9*i);
23:       }
24:
25:       sort( v1.begin(), v1.end() );
26:       cout << "Sorted vector v1 = " ;
27:       for( It=v1.begin(); It != v1.end(); It++ )
28:            cout << *It << " ";
29:       cout << endl;
30:
31:       sort( v1.begin(), v1.end(), UDgreater);
32:       cout << "Sorted vector v1 = " ;
33:       for( It=v1.begin(); It != v1.end(); It++ )
34:            cout << *It << " ";
35:       cout << "\n\n" << endl;
36:
37:       return 0;
38:  }
```

У лістингу 10.9 в рядках 7-12 вводиться функція порівняння `UDgreater()`, яка порівнює модулі двох цілих чисел.

В тілі функції `main()` створюється об'єкт-вектор `vector<int> v1` та ітератор для нього `It`. В рядках 20-23 вектор заповнюється значеннями `2*i*i - 9*i`, залежними від індексу елемента `i`.

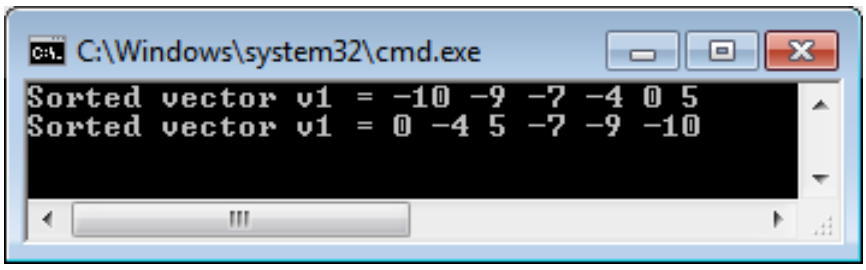


Рис. 39

У рядку 25 відбувається сортування елементів вектора за замовчуванням – за зростанням значень елементів. Вміст відсортованого вектора виводиться

на консоль за допомогою блоку коду в рядках 27-28.

Далі елементи вектора знову впорядковуються, при чому для цього використовується функція порівняння `UDgreater()` (рядок 31). Вміст вектора, відсортованого ще раз, вже з використанням `UDgreater()`, виводиться на консоль за допомогою блоку коду в рядках 33-34.

Результати роботи програми з лістингу 10.9 показані на рис. 39.

Завдання

Варіант 1

1) Створити два класи: клас комплексних чисел (що містить поля *x* та *y* типу `double` для дійсної та уявної частини) та клас дробів (поля цілого типу *m* та *n* для чисельника та знаменника).
2) Задати конструктор та деструктор кожного класу. В конструкторах передбачити генерацію виключення у випадку коли $x = y = 0$ (не визначена фаза комплексного числа) та $n = 0$ (ділення на нуль).
3) Перевантажити для кожного з цих класів операцію додавання `+` для додавання комплексних чисел та дробів, відповідно.
4) Створити шаблон функції, яка буде додавати два однотипних об'єкти одного із вказаних вище класів.
5) Створити декілька об'єктів кожного з класів та продемонструвати для них виконання операції додавання, використовуючи як перевантажені оператори, так і шаблонну функцію.
6) Створити масив `std::vector` з 5 екземплярами класу комплексних чисел та масив `std::vector` з 5 екземплярами класу дробів. Заповнити масиви довільними даними.
7) Вивести на екран масив комплексних чисел, відсортований за зростанням дійсної частини, та масив дробів, відсортований за зростанням чисельника.

Варіант 2

1) Створити шаблонний клас комплексних чисел (що містить поля довільного типу *x* та *y* для дійсної та уявної частин). В класі описати конструктор, деструктор, методи обрахунку модуля та фази комплексного числа, методи для присвоювання значень полям *x* та *y* та метод виведення числа на екран у форматі $x + iy$.
2) В конструкторі передбачити генерацію виключення у випадку, коли $x = y = 0$ (не визначена фаза комплексного числа).
3) Створити декілька об'єктів вказаного класу, в яких *x* та *y* представлені

числами типу `int`, `double` та `float`.

4) Описати шаблон функції, що виконує множення комплексного числа на дійсне число довільного типу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів класу комплексних чисел, в яких x та y представлені числами типу `int` та `float`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного з контейнерів з попереднього пункту, відсортований за зростанням модуля комплексного числа.

Варіант 3

1) Створити два класи: клас комплексних чисел (що містить поля x та y типу `float` для дійсної та уявної частин) та клас дробів (поля цілого типу m та n для чисельника та знаменника).
2) Задати конструктор та деструктор кожного класу. В конструкторах передбачити генерацію виключення у випадку коли $x = y = 0$ (не визначена фаза комплексного числа) та $n = 0$ (ділення на нуль).
3) Перевантажити для кожного з цих класів операцію множення `*` для множення комплексних чисел та дробів, відповідно.
4) Створити шаблон функції, яка буде множити два однотипних об'єкти одного із вказаних вище класів.
5) Створити декілька об'єктів кожного з класів та продемонструвати для них виконання операції множення, використовуючи як перевантажені оператори, так і шаблонну функцію.
6) Створити масив `std::vector` з 5 екземплярами класу комплексних чисел та масив `std::vector` з 5 екземплярами класу дробів. Заповнити масиви довільними даними.
7) Вивести на екран масив комплексних чисел, відсортований за зменшенням уявної частини, та масив дробів, відсортований за зростанням знаменника.

Варіант 4

1) Створити шаблонний клас комплексних чисел (що містить поля довільного типу x та y для дійсної та уявної частин). В класі описати конструктор, деструктор, методи обрахунку модуля та фази комплексного числа, методи для присвоювання значень полям x та y та метод для виведення числа на екран у форматі $x + iy$.
2) В конструкторі передбачити генерацію виключення у випадку, коли $x = y = 0$ (не визначена фаза комплексного числа).
3) Створити декілька об'єктів вказаного класу, в яких x та y представлені

числами типу `int`, `double` та `float`.

4) Описати шаблон функції додавання до комплексного числа дійсного/цілого числа довільного типу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів класу комплексних чисел, в яких x та y представлені числами типу `short` та `double`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за зменшенням аргументу комплексного числа.

Варіант 5

1) Створити два класи: клас дробів (що містить поля цілого типу m та n для чисельника та знаменника) та клас тривимірних точок (що містить три декартові координати x, y та z).
2) Задати конструктор та деструктор кожного класу. В конструкторах передбачити генерацію виключення у випадку коли $x = y = z = 0$ (точка не повинна співпадати з початком координат) та $n = 0$ (ділення на нуль).
3) Перевантажити для кожного з цих класів операцію виведення даних в потік `<<`, так щоб дані дробу виводились у формі відношення m/n, а дані точки – у вигляді (x, y, z).
4) Створити шаблон функції, яка отримавши аргумент у вигляді контейнера `std::vector`, який містить об'єкти одного із вказаних класів, буде виводити на екран його вміст за описаним вище форматом.
5) Створити два масиви типу `std::vector` з 5 екземплярів класу дробів та тривимірних точок кожний і заповнити їх довільними даними.
6) Відсортувати елементи масивів за зростанням модуля дробу та відстані від точки до початку координат.
7) Вивести на екран відсортовані масиви за допомогою функції, реалізованої в п. 4.

Варіант 6

1) Створити шаблонний клас комплексних чисел (що містить поля довільного типу x та y для дійсної та уявної частин). В класі описати конструктор, деструктор, методи обрахунку модуля та фази комплексного числа, методи для присвоювання значень полям x та y та метод для виведення числа на екран у форматі $x + iy$.
2) В конструкторі передбачити генерацію виключення у випадку, коли $x = y = 0$ (не визначена фаза комплексного числа).
3) Створити декілька об'єктів вказаного класу, в яких x та y представлені числами типу `short`, `int` та `float`.

4) Описати шаблон функції, яка підносить комплексне число до цілого ступеня довільного типу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті, і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів класу комплексних чисел, в яких *x* та *y* представлені числами типу `short` та `float`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за збільшенням дійсної частини комплексного числа.

Варіант 7

1) Створити два класи: клас комплексних чисел (що містить поля дійсного типу *x* та *y* для дійсної та уявної частин) та клас тривимірних векторів (що містить три декартові компоненти *x*, *y* та *z*).
2) Задати конструктор та деструктор кожного класу. В конструкторах передбачити генерацію виключення у випадку коли $x = y = z = 0$ (вектор не повинен бути нульовим) та $x = y = 0$ (не визначена фаза комплексного числа).
3) Визначити шаблон функції, яка отримавши як аргумент об'єкт одного із вказаних вище класів, обчислює його модуль (модуль комплексного числа або модуль вектора, відповідно). Результатом роботи функції має бути дійсне число типу `float`.
4) Створити два списки типу `std::list` з 5 екземплярами класу комплексних чисел та тривимірних векторів кожний і заповнити їх довільними даними.
5) Відсортувати елементи контейнерів за зменшенням модуля комплексного числа та модуля вектора, відповідно.
6) Вивести на екран елементи відсортованих масивів у форматі *x* + i*y*, *модуль комплексного числа* та (*x*, *y*, *z*), *модуль вектора*.

Варіант 8

1) Створити шаблонний клас для опису інтегралів від степеневої функції x^n (що містить поля довільного типу *x* та *n* для змінної інтегрування та степені змінної). В класі описати конструктор, деструктор, методи для обрахунку невизначеного інтеграла та визначеного інтеграла із заданими межами, методи для присвоювання значень полям *x* та *n* та метод для виведення результату інтегрування у вигляді:
   Integral(y, 4.5) = (y^5.5)/5.5,
   Ranges(1, 2) Integral(x, 2) = 7/3
2) В конструкторі передбачити генерацію виключення у випадку коли *n* = –1 (інша формула для розрахунку інтегралу).

3) Створити декілька об'єктів вказаного класу, в яких *x* та *n* представлені змінними типу `char` та `int`, `double` та `float`, відповідно.
4) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 3 об'єкти вказаного класу кожний, в яких *x* та *n* представлені типом `char` та `float`. Заповнити контейнери випадковими даними.
5) Вивести на екран вміст кожного з контейнерів з попереднього пункту, відсортований за збільшенням результату інтегрування при інтегруванні в межах від 1 до 2.

Варіант 9

1) Створити шаблонний клас похідних однієї змінної від степеневої функції x^n (що містить поля довільного типу *x* та *n* для змінної та її степені). В класі описати конструктор, деструктор, метод для обрахунку похідної, методи для присвоювання значень полям *x* та *n* та метод для виведення результату диференціювання у вигляді:
   Derive (z, 2.5) = 2.5*(z^1.5)
2) В конструкторі передбачити генерацію виключення у випадку, коли $n = 0$ (результат не залежить від *x*).
3) Створити декілька об'єктів вказаного класу, в яких *x* та *n* мають тип `char` та `int`, `double` та `float`, відповідно.
4) Описати шаблон функції яка знаходить значення похідної в заданій точці $x = x_0$ (x_0 може мати тип `int`, `double`, `float`). Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 3 об'єкти вказаного класу, в яких *x* та *n* представлені типом `char` та `double`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за зменшенням значення похідної в точці $x_0 = 1$.

Варіант 10

1) Створити шаблонний клас для опису квадратної матриці *A* розміром 2×2 (що містить поля довільного типу a_{ij}, де $i,j = 1,2$). В класі описати конструктор, деструктор, метод для обрахунку детермінанту det(A), методи для присвоювання значень полям a_{ij}, метод для множення матриці на число та метод для виведення матриці на екран у вигляді:
   1 –2
   3 2
2) В конструкторі передбачити генерацію виключення у випадку, коли детермінант дорівнює нулю (вивести текстове попередження про це).
3) Створити декілька об'єктів вказаного класу, в яких a_{ij}, представлені числами типу `int`, `double`, `float`.

4) Описати шаблон функції, яка знаходить обернену матрицю до даної. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран. Нагадаємо, що обернена матриця в цьому випадку має вигляд:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів вказаного класу, в яких a_{ij}, представлені числами типу `float`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за збільшенням величини детермінанту.

Варіант 11

1) Створити шаблонний клас `Student`, що містить поля з інформацією про прізвище, ім'я, по-батькові (рядок символів), стать (типу `char`), вік і курс (узагальнені типи) та середній бал (тип `float`). В класі описати конструктор, деструктор, методи для присвоювання значень всім полям класу та метод для виведення на екран повної інформації про студента у довільному форматі.
2) Передбачити генерацію виключення у випадку, коли поле віку менше 0 або більше 100 та коли в поле статі задається не одним із символів Ч або Ж.
3) Створити декілька об'єктів вказаного класу, в яких поля віку і курсу представлені числами типу `int`, `double` та `float`.
4) Описати шаблон функції для розрахунку середнього віку кількох студентів. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 4 об'єкти вказаного класу, в яких поля віку і курсу представлені числами типу `unsigned int` та `float`, відповідно. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за зростанням номера курсу.

Варіант 12

1) Створити клас раціональних дробів, що містить поля цілого знакового типу *m* та *n* для чисельника та знаменника.
2) Задати конструктор та деструктор. В конструкторі передбачити генерацію виключення у випадку коли *n* = 0 (ділення на нуль).

3) Перевантажити для цього класу оператор виведення даних в потік << так, щоб дріб виводився у форматі відношення m/n.
4) Перевантажити для класу оператори порівняння < та >.
5) Записати шаблон функції, яка буде визначати та повертати максимальне із двох чисел, заданих в списку її аргументів. Продемонструвати роботу цієї функції для випадку аргументів із переліку стандартних числових типів даних та для випадку раціональних дробів.
6) Створити масив типу `std::vector`, що містить 5 екземплярів класу раціональних дробів. Заповнити контейнер випадковими даними (поля m та n мають бути як додатними, так і від'ємними).
7) Вивести на екран масив дробів, відсортований за зростанням модуля дробу.

Варіант 13

1) Створити шаблонний клас `Student`, що містить поля з інформацією про прізвище, ім'я, по-батькові (рядок символів), стать (типу `bool`), вік і курс (узагальнені типи) та середній бал (тип `double`) . В класі описати конструктор, деструктор, методи для присвоювання значень всім полям класу та метод для виведення на екран повної інформації про студента у довільному форматі.
2) Передбачити генерацію виключення у випадку, коли поле курсу менше 0 або більше 6.
3) Створити декілька об'єктів вказаного класу, в яких поля віку і курсу представлені числами типу `long int`, `double` та `float`.
4) Описати шаблон функції, яка отримавши масив об'єктів типу `Student` та номер курсу, виведе на екран дані про всіх студентів заданого курсу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів вказаного класу, в яких поля віку і курсу представлені числами типу `double` та `char`, відповідно. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за зменшенням віку студента.

Варіант 14

1) Створити шаблонний клас для опису інтегралів від квадратичної степеневої функції $f(x) = ax^2 + bx + c$ (що містить поля довільного типу a, b, c для коефіцієнтів полінома і два поля довільного типу L та H для верхньої та нижньої меж інтегрування). В класі описати конструктор, деструктор, методи для присвоювання значень всім полям, метод для обрахунку

визначеного інтеграла із заданими межами від відповідного поліному.

2) В конструкторі передбачити генерацію виключення у випадку коли $L > H$ (нижня межа має бути меншою за верхню) та якщо $a = 0$.
3) Створити декілька об'єктів вказаного класу, в яких поля *a, b, c, L* та *H* представлені змінними типу `int`, `double`, `float` у довільних комбінаціях.
4) Створити контейнери типу `std::vector` та `std::list`, що містять по 4 об'єкти вказаного класу, в яких всі поля представлені однаковим типом `int` або `float`. Заповнити контейнери випадковими даними.
5) Вивести на екран вміст першого контейнера з попереднього пункту, відсортований за збільшенням значення інтегралу, та вміст другого, відсортований за зменшенням значення інтегралу.

Варіант 15

1) Створити шаблонний клас для опису квадратної матриці *A* розміром 3×3 (що містить 9 елементів узагальненого типу). В класі описати конструктор, деструктор, методи для присвоювання значень полям та метод для виведення матриці на екран у вигляді:

```
1    2    3
4    5    6
7    8    9
```

2) В конструкторі передбачити генерацію виключення у випадку коли всі елементи матриці одночасно дорівнюють нулю.
3) Перевантажити для цього класу операції додавання + та віднімання −. Дві матриці додаються або віднімаються поелементно.
4) Перевантажити для даного класу оператор множення на число (перший аргумент має бути матрицею, інший – змінною числового типу). При множенні матриці на число, на нього множиться кожен елемент матриці окремо.
5) Створити декілька об'єктів вказаного класу, в яких елементи матриці представлені числами типу `int` або `double`. Продемонструвати на їх прикладі роботу перевантажених операторів додавання та віднімання матриць, множення матриці на число.
6) Створити контейнери типу `std::vector` та `std::list`, що містять 3 об'єкти вказаного класу, в яких елементи матриць представлені числами типу `float` або `unsigned int`. Заповнити контейнери випадковими даними.
7) Вивести на екран вміст кожного з контейнерів з попереднього пункту, відсортований за збільшенням величини максимального із елементів матриці.

Варіант 16

1) Створити шаблонний клас для опису двовимірного вектора (що містить поля довільного типу *x* та *y* для відповідних проекції в декартовій системі координат). В класі описати конструктор, деструктор, метод для обрахунку модуля вектора та визначення одиничного вектора (вектор паралельний даному, але з модулем 1), методи для присвоювання значень полям *x* та *y* та метод для виведення вектора на екран у форматі *x**i + y*j.
2) В конструкторі передбачити генерацію виключення у випадку коли *x* = *y* = 0 (нуль-вектор).
3) Створити декілька об'єктів вказаного класу, в яких *x* та *y* представлені числами типу `int`, `double` та `float`.
4) Описати шаблон функції для множення вектора на число довільного типу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів вказаного класу, в яких *x* та *y* представлені числами типу `int` та `float`. Заповнити контейнери випадковими даними.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за зменшенням модуля вектора.

Варіант 17

1) Створити шаблонний клас для опису тривимірного вектора (що містить поля довільного типу *x*, *y* та *z* для відповідних проекції в декартовій системі координат). В класі описати конструктор, деструктор, метод для обрахунку модуля вектора та визначення одиничного вектора (вектор паралельний даному, але з модулем 1), методи для присвоювання значень полям *x*, *y* та *z* та метод для виведення вектора на екран у форматі *x**i + y*j + z*k.
2) В конструкторі передбачити генерацію виключення у випадку коли *x* = *y* = *z* = 0 (нуль-вектор).
3) Створити декілька об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `int`, `double` та `float`.
4) Описати шаблон функції для множення вектора на число довільного типу. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::vector` та `std::list`, кожен з яких містить по 5 об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `int` та `float`. Заповнити контейнери випадковими даними в діапазоні значень від 0 до 1000.
6) Вивести на екран вміст кожного з контейнерів з попереднього пункту,

відсортований за збільшенням модуля вектора.

Варіант 18

1) Створити шаблонний клас для опису тривимірного вектора (що містить поля довільного типу *x*, *y* та *z* для відповідних проекції в декартовій системі координат). В класі описати конструктор, деструктор, метод для обрахунку модуля вектора та визначення одиничного вектора (вектор паралельний даному, але з модулем 1), методи для присвоювання значень полям *x*, *y* та *z* та метод для виведення вектора на екран у форматі *x**i + y*j + z*k.
2) В конструкторі передбачити генерацію виключення у випадку коли *x* = *y* = *z* = 0 (нуль-вектор).
3) Створити декілька об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `short int`, `long int` та `float`.
4) Описати шаблон функції для обрахунку скалярного добутку двох векторів (довільних типів). Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::list` та `std::vector`, кожен з яких містить по 5 об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `int` та `double`. Заповнити контейнери випадковими даними в діапазоні значень від –100 до +100.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за збільшенням величини скалярного добутку векторів. Перший вектор-множник має обиратись з контейнера довільним чином.

Варіант 19

1) Створити шаблонний клас для опису тривимірного вектора (що містить поля довільного типу *x*, *y* та *z* для відповідних проекції в декартовій системі координат). В класі описати конструктор, деструктор, метод для обрахунку модуля вектора та визначення рівняння площини (символьний рядок), ортогональної до вектора, що проходить через точку (0,0,0) [рівняння площини має вигляд *x**X+*y**Y+*z**Z = 0], методи для присвоювання значень полям *x*, *y* та *z* та метод для виведення вектора на екран у форматі *x**i + y*j + z*k.
2) В конструкторі передбачити генерацію виключення у випадку коли *x* = *y* = *z* = 0 (нуль-вектор).
3) Створити декілька об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `signed char`, `int` та `float`.
4) Описати шаблон функції для обрахунку векторного добутку двох векторів (довільних типів). Застосувати цю функцію до декількох різнотипних

об'єктів, створених в попередньому пункті і довести її працездатність. Результати роботи функції вивести на екран.

5) Створити контейнери типу `std::list` та `std::vector`, кожен з яких містить по 5 об'єктів вказаного класу, в яких *x, y* та *z* представлені числами типу `int` та `double`. Заповнити контейнери випадковими даними в діапазоні значень від 0 до 100.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за збільшенням модуля векторного добутку вектора зі сталим вектором (1,1,1).

Варіант 20

1) Створити шаблонний клас для опису тригонометричних функцій (містить поле кута `phi`, яке приймає цілі значення, якщо кут задано в градусах, та дійсні значення, якщо кут задано в радіанах, та поле амплітуди `A` довільного типу). В класі описати конструктор, деструктор, методи для обрахунку косинуса та синуса кута, метод для присвоювання значень полям `phi` та `A` та метод для виведення інформації на екран у форматі: `A*cos(phi) + A*sin(phi)`.
2) В конструкторі передбачити генерацію виключення у випадку коли `A` та `phi` дорівнюють нулю.
3) Створити декілька об'єктів вказаного класу, в яких `A`, `phi` представлені числами типу `int` та `float`.
4) Описати шаблон функції для розрахунку значення виразу `A*cos(phi) + A*sin(phi)`. Застосувати цю функцію до декількох різнотипних об'єктів, створених в попередньому пункті і довести її працездатність. Якщо `phi` має тип `int`, вважати, що кут задається в градусах, якщо `float` – в радіанах. Результати роботи функції вивести на екран.
5) Створити контейнери типу `std::list` та `std::vector`, кожен з яких містить по 5 об'єктів вказаного класу, в яких `A`, `phi` представлені числами типу `int` та `float`. Заповнити контейнери випадковими даними в діапазоні значень від 0 до 180.
6) Вивести на екран вміст кожного із контейнерів з попереднього пункту, відсортований за збільшенням значення виразу `A*cos(phi) + A*sin(phi)`.

Контрольні запитання

1. Що таке шаблони? Для чого вони потрібні? Наведіть приклади. Які переваги притаманні шаблонам, які їх недоліки?
2. Чим відрізняються шаблони класів і шаблони функцій? Синтаксис запису

шаблонів.

3. Що таке винятки (виключення), для чого вони застосовуються, за якими принципами реалізується їх обробка? Наведіть приклади.
4. Який синтаксис обробки винятків (виключень) у мові C++? Що таке блоки `try`? Яку функцію в програмі виконує блок `catch`? Наведіть приклади.
5. Чи важливий порядок запису блоків `catch` після блоку `try`? Якщо відповідь позитивна, поясніть за яким принципом слід записувати блоки `catch`.
6. Яким чином можна генерувати виключення в програмах на C++? Наведіть приклади.
7. Що таке ієрархія винятків (виключень)? На яких принципах вона ґрунтується, для чого застосовується?
8. Що таке бібліотека STL? Які принципи покладені в її основу? Які її ключові елементи?
9. Що таке ітератор? Чим ітератори відрізняються від вказівників? Чим вони схожі на вказівники? Наведіть приклади.
10. Охарактеризуйте можливості контейнерного класу `std::vector` з бібліотеки STL. Наведіть приклади використання цього класу.
11. Охарактеризуйте можливості контейнерного класу `std::list` з бібліотеки STL. Наведіть приклади використання цього класу.
12. Що таке алгоритми STL? Наведіть приклад використання алгоритму `sort()` для сортування даних.

Рекомендована література

1. С++. Основи Програмування. Теорія та практика: підручник / О.Г. Трофименко, Ю.В. Прокоп, І.Г. Швайко, Л.М. Буката, Л.А. Косирева, Ю. Г. Леонов, В. В. Ясинський. – Фенікс, 2010.
2. С у задачах і прикладах: навчальний посібник із дисципліни «Інформатика та програмування» / А.П. Креневич, О.В. Обвінцев. – К.: Видавничо-поліграфічний центр «Київський університет», 2011. – 208 с.
3. Вступ до програмування мовою С++. Організація обчислень: навчальний посібник / Ю.А. Бєлов, Т.О. Карнаух, Ю.В, Коваль, А.Б. Ставровський. – К.: Видавничо-поліграфічний центр «Київський університет», 2012. – 175 с.
4. Програмування мовами С та С++: навч. посіб / Д.Д. Татарчук, Ю.В. Діденко. – К. НТУУ «КПІ», 2012. – 112 с.
5. Програмування на С++ в прикладах і задачах: навч. посібник / О. Васильєв. – К: Ліра, 2020. – 381с.
6. Методичні рекомендації з курсу «Мова програмування С++» / В.А. Бородін. – К.: КНУ, 2021. – 101 с.

Н а в ч а л ь н и й п о с і б н и к

ПРОКОПЕНКО Олександр Володимирович
ПОПОВ Максим Олександрович
ЧУМАК Григорій Леонідович

МОВА ПРОГРАМУВАННЯ С/С++. ПРАКТИКУМ

Підписано до друку 04.06.2024. Формат 60×84/16.
Гарнітура Times New Roman. Ум.-друк. арк. 23,4.

Навчально-науковий інститут високих технологій
Київського національного університету імені Тараса Шевченка

375