

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

КОВАЛЮК Т.В.
ПЕТРІВСЬКИЙ В.Я.

ГАРНИЙ СТИЛЬ ПРОГРАМУВАННЯ МОВАМИ C, C++, C#
Інструкція до написання програмного коду

Київ 2025

Рецензенти:

М.І. Цюцюра, д.т.н., професор кафедри інженерії програмного забезпечення та кібербезпеки Державного торговельно-економічного університету

О. Є. Іларіонов, к.т.н., доцент, завідувач кафедри інтелектуальних технологій Київського національного університету імені Тараса Шевченка

Рекомендовано до публікації кафедрою програмних систем і технологій, протокол № 1 від «29» серпня 2024 р.

Рекомендовано до публікації Вченою радою факультету інформаційних технологій, протокол № 6 від «13» січня 2025 р.

Ковалюк Тетяна Володимирівна, Петрівський Володимир Ярославович

Гарний стиль програмування мовами C, C++, C#. Інструкція до написання програмного коду [Електронний ресурс]. Кафедра програмних систем і технологій / Т.В. Ковалюк, В.Я. Петрівський. – К.: КНУ імені Тараса Шевченка, 2025. – 59 с.

Гарний стиль написання програмного коду є невід’ємною частиною процесу розробки програмного забезпечення. У праці представлено інструкції дотримання гарного стилю програмування при написанні програмного коду мовами програмування C, C++, C#. Інструкція призначена для студентів 1 курсу освітнього ступеня «бакалавр», спеціальності «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення», навчальних дисциплін «Основи програмування» та «Вступ до об’єктно-орієнтованого програмування».

Публікується в авторській редакції.

Електронна версія цього видання опублікована на сайті кафедри програмних систем і технологій за адресою: <https://pst.knu.ua/library> (дата публікації «30» січня 2025 р.)

© Ковалюк Т.В., Петрівський В.Я., 2025 р.

Зміст

Вступ	4
Частина 1 Процедурно-орієнтований стиль програмування	4
1 Стиль проєктування (Design Style).....	4
1.1 Принципи процедурного та структурного проєктування	4
1.2 Вимоги до проєктування процедур (функцій)	6
1.3 Технологія розробки процедурно-орієнтованої програми	7
1.4 Ознаки правильно спроектованої процедурно-орієнтованої програми	7
Резюме.....	8
Контрольні запитання.....	9
Контрольні завдання.....	9
2 Стиль кодування	9
2.1 Визначення імен	9
2.2 Використання змінних і констант (літералів)	17
2.3 Вирази	22
2.4 Функції та оператори	23
Резюме.....	27
Контрольні запитання.....	28
Контрольні завдання.....	28
3 Стиль документування та форматування коду	29
3.1 Повідомлення про авторські права	29
3.2 Коментування	29
3.3 Форматування	35
Резюме.....	42
Контрольні запитання.....	42
Контрольні завдання.....	43
Частина 2. Об'єктно-орієнтований стиль програмування мовою C#	45
4 Проєктування класів C#	45
4.1 Види класів	45
4.2 Принципи об'єктно-орієнтованого програмування	46
4.3 Структура класів	46
4.4 Мінімальні або компактні класи	49
4.5 Принцип єдиної відповідальності	49
5 Програмування класів в C#.....	50
5.1 Правила визначення імен	50
5.2 Форматування коду C#	52
5.3 Коментування коду C#	54
Резюме.....	55
Контрольні запитання.....	56
Контрольні завдання.....	56
Використані джерела	59

Вступ

Стиль програмування — це набір правил програмування, яких дотримуються всі програмісти для написання коду. Коли кілька програмістів працюють над одним програмним проєктом, їм часто доводиться працювати з програмним кодом, написаним іншим розробником. Це стає виснажливим або часом неможливим, якщо всі розробники не дотримуються стандартного стилю програмування для кодування програми.

Стиль програмування передбачає використання імен функцій і змінних, що відповідають поставленій задачі, правильного розміщення відступів, коментування коду для зручності читача та загального подання коду. Це робить програмний код читабельним і зрозумілим для всіх, що, у свою чергу, полегшує налагодження та виправлення помилок. Правильний стиль кодування полегшує документування та оновлення коду.

У першій частині методичних рекомендацій розглядатимемо питання щодо гарного стилю процедурно-орієнтованого програмування мовами C та C++. Друга частини присвячена питанням гарного стилю об'єктно-орієнтованого програмування мовою C#.

Методичні рекомендації з гарного стилю програмування мовами C, C++, C# призначені для формування у студентів вміння писати чистий код, вимоги до якого сформульовані в багатьох джерелах, названих C++ Coding Standard, або C# Coding Standard. Цільова аудиторія цих рекомендацій – студенти, викладачі та усі користувачі, які хочуть підвищити свою кваліфікацію як програміста. Особливо важливим є формування у студентів молодших курсів розуміння і навички, як писати хороший код і як перетворити поганий код на хороший.

Рекомендації містять велику кількість реальних прикладів коду. Їх аналіз буде особливо корисним для студентів, що вивчають дисципліни «Основи програмування» і «Вступ до об'єктно-орієнтованого програмування».

Частина 1 Процедурно-орієнтований стиль програмування

1 Стиль проєктування (Design Style)

Проєктування програмного забезпечення — це процес визначення програмних методів, функцій, об'єктів, а також загальної структури та взаємодії коду таким чином, щоб отримана функціональність задовольняла вимогам користувачів.

1.1 Принципи процедурного та структурного проєктування

У цьому розділі розглядаються загальні поняття процедурно-орієнтованого та структурного програмування та такі принципи як «розділяй і володарюй», модульність, програмування зверху вниз або спадна розробка, принцип простоти, принцип повторного використання коду.

Процедурно-орієнтоване програмування — це парадигма програмування, що походить від імперативного програмування, заснованого на концепції викликів процедур. На етапі процедурно-орієнтованого проєктування здійснюється алгоритмічна декомпозиція задачі на окремі логічні блоки. Логіка програми записується в окремих процедурах (функціях або методах) і містить послідовність обчислювальних кроків, які необхідно виконати. Будь-яка процедура може викликатися в будь-який момент виконання програми, в тому числі іншими процедурами або самою собою. Код процедури можна повторно використовувати в різних частинах програми, де це не заборонено синтаксисом мови програмування, без необхідності його копіювання. Отже, програма, по суті, є послідовністю викликів процедур, складні процедури можна декомпонувати на процедури меншої складності за принципом: одна процедура – один логічний блок.

У процедурному програмуванні дані та процедури розділені, а не об'єднані разом. Це суттєва відмінність від об'єктно-орієнтованого програмування, де дані та методи інкапсулюються разом в об'єкти.

Процедурне програмування часто використовує глобальні дані, які можуть бути доступні та змінені будь-якою частиною програми. Це може бути як перевагою, так і недоліком, оскільки може призвести до небажаних побічних ефектів, якщо його ретельно не контролювати.

Переваги використання процедурного програмування:

- простота;
- ефективність;
- портативність.

Недоліки використання процедурного програмування:

- обмежена гнучкість і складність розширення коду;
- складності супроводу і модифікації великих програм.

Структурне проєктування програми передбачає алгоритмічну низхідну декомпозицію, модульну структуру програми, структурне кодування (програмування).

Структурне програмування – це методологічний підхід до написання програмного коду, який представляє програму у вигляді структури з набору блоків, розташованих в ієрархічній послідовності.

Три принципи структурного програмування [1]:

- алгоритмічна низхідна декомпозиція означає, що поставлене завдання покроково деталізується, у напрямку від самого верхнього рівня, вниз до дрібних деталей. Цей метод дає змогу створити чітку структуру програми. Цей принцип називається «програмування зверху вниз або спадна розробка». Результатом алгоритмічної декомпозиції задачі є HIPO (Hierarchical Input Process Output) діаграма (рис.1).

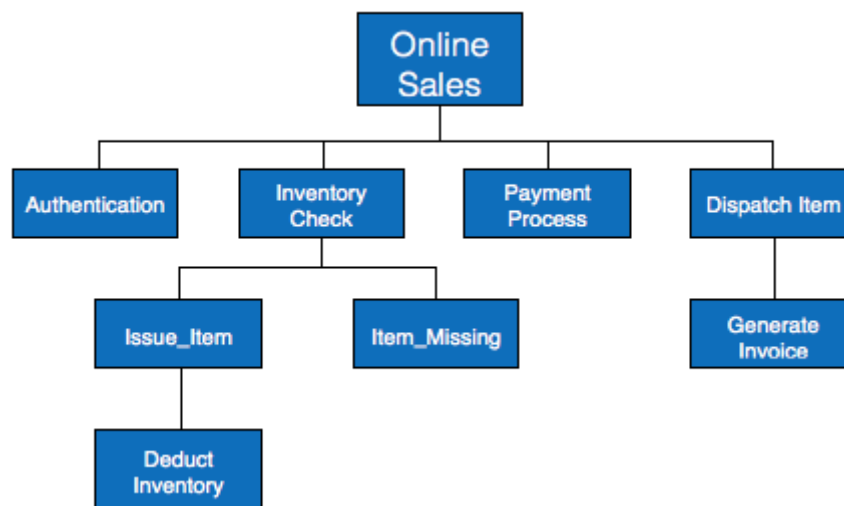


Рисунок 1. – HIPO діаграма системи онлайн продажів

- модульна структура програми означає, що в результаті декомпозиції програма розбивається на модулі, прості елементи.
- під час структурного кодування використовують три структури керування: послідовність, розгалуження, повторення, які є базовими логічними структурами (рис.2):

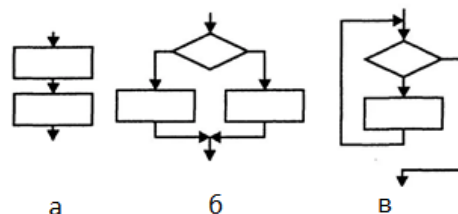


Рисунок 2. – Базові логічні структури програмування:
а) – послідовність, б) – розгалуження, в) - повторення

Переваги структурного програмування:

- код легко читати;
- код більш простіший;
- код надійніший, що означає зменшення помилок або збоїв;
- код простіше обслуговувати, його легше змінити або оновити в міру необхідності.

Недоліки структурного програмування:

- непропорційне зростання складності програми у разі збільшення об'єму коду програми;
- складність створення паралельних програм.

Семантика принципів процедурного та структурного програмування та проектування.

Принцип «розділяй і володарюй» означає поділ великої задачі на менші частини, як наслідок просте вирішення малих підзадач, що в свою чергу породжує вирішення великої задачі. Реалізація принципу «розділяй і володарюй» здійснюється через застосування процедур (функцій).

Принцип повторного використання коду (DRY – don't repeat yourself – «не повторюйся») означає, що *«будь-яка інформація повинна мати єдине, однозначне, авторитетне представлення в системі»* (Dave Thomas). Задача принципу — позбутися дублювання логіки, що сильно спростить майбутню підтримку коду, адже за необхідності щось змінити достатньо буде це зробити лише в одному місці програми.

Принцип модульності є наслідком принципу «розділяй і володарюй» і означає, що програма розбивається на логічно незалежні частини (модулі), які характеризуються одним входом і одним виходом, логічною та програмною незалежністю модуля в межах інтерфейсу програми і структури даних (заміна модуля на аналогічний не повинна впливати на всю програму), кожний модуль відповідає лише одній задачі, модуль не повинен зберігати історію своїх викликів, інакше може виникати побічний ефект.

Принцип програмування «зверху вниз» або спадна розробка передбачає послідовну деталізацію проєкту від модулів верхнього рівня до модулів нижнього рівня. Для кожного модуля визначають алгоритм розв'язку задачі, області допустимих і можливих вихідних значень змінних.

Принцип простоти (KISS – keep it short and simple — «роби коротше і простіше») означає, що системи працюють краще, якщо вони прості в користуванні. Виходячи з цього, простота повинна бути головною метою в області дизайну і потрібно намагатися уникати непотрібних складнощів під час проєктування.

Процедурне програмування є підмножиною структурного програмування. Зв'язок процедурного та структурного програмування поданий на рис.3. Імперативне програмування є парадигмою програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми. Подібно до того, як з допомогою наказового способу в мовознавстві перелічується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп'ютеру для виконання.

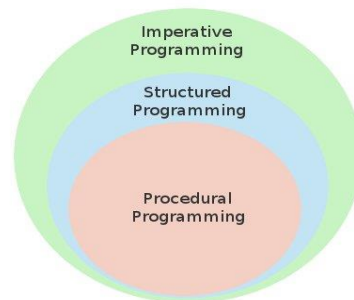


Рисунок 3. – Зв'язок процедурного та структурного програмування

1.2 Вимоги до проєктування процедур (функцій)

У процесі проєктування функцій потрібно дотримуватися таких правил:

- функція має повністю виконувати чітко поставлене завдання;
- функція не повинна брати на себе занадто багато роботи;
- функція не повинна бути пов'язаною з іншими функціями безцільно;
- функція має зберігати дані максимально стисло;
- функція повинна допомагати розпізнати та розділити структуру програми на окремі відносно незалежні блоки;

- функція повинна допомагати позбутися надлишків, які інакше були б у програмі;
- функція використовує параметри, щоб передати дані з функції у випадку повернення декількох значень;
- функція не повинна використовувати параметри без необхідності;
- функція `main()` є коротким описом усієї програми, є точкою входу до програми і виконує роль диспетчера, що викликає інші функції програми;
- потрібно уникати «ланцюгових» викликів функцій, коли безліч функцій викликають один за одним по ланцюжку, не повертаючи значення в `main()`;
- потрібно уникати довгих функцій і глибокої їх вкладеності.

1.3 Технологія розробки процедурно-орієнтованої програми

Розробка програми ведеться покроково, методом «зверху вниз» за принципом «розділай і володарюй».

Крок 1. Спочатку пишеться текст основної програми, в якому, замість кожного логічного фрагмента тексту вставляється виклик підпрограми, яка виконуватиме цей фрагмент. Замість працюючих підпрограм в програму вставляються «заглушки», які нічого не роблять. Отримана програма перевіряється і налаштовується.

Крок 2. Після того, як програміст переконається, що підпрограми викликаються в правильній послідовності (тобто загальна структура програми вірна), у підпрограми-заглушки вставляється реально працюючий код, розробка якого ведеться тим же методом, що і основної програми. Розробка програми закінчується тоді, коли не залишиться жодної «заглушки».

Така послідовність гарантує, що на кожному етапі розробки програміст одночасно має справу з зрозумілим йому безліччю фрагментів, і може бути упевнений, що загальна структура всіх вищих рівнів програми вірна.

Крок 3. При супроводі і внесенні змін в програму з'ясовується, в які саме процедури потрібно внести зміни, і вони вносяться, не зачіпаючи частини програми, безпосередньо не пов'язані з ними. Це дозволяє гарантувати, що при внесенні змін і виправленні помилок не вийде з ладу якась частина програми, що знаходиться в даний момент поза зоною уваги програміста.

1.4 Ознаки правильно спроектованої процедурно-орієнтованої програми

Програма, що відповідає принципам процедурного програмування і розроблена відповідно до принципів структурного проектування, характеризується такими чинниками:

Одна сутність (процедура, функція) – одна задача. Якщо під сутністю розуміти процедуру (функцію), кожна сутність у програмі повинна мати одне ясне призначення. Сутність з різнорідними призначеннями зазвичай важко використовувати, оскільки вона є чимось більшим, ніж просто сума рішень, складнощів і помилок складових її частин. Така сутність більше за розміром (часто без особливих причин) і складніше у застосуванні та повторному використанні. Найчастіше такі сутності мають дуже убогий інтерфейс для кожного зі своїх окремих призначень, оскільки часткове перекриття різних областей функціональності призводить до неможливості чіткої реалізації у кожному їх. Переважно слід використовувати короткі функції з чітко зазначеним єдиним призначенням для вирішення одного конкретного завдання, і компактні модулі з чітко окресленими межами [2].

Коректність, простота та ясність. Ця ознака свідчить про те, що коректність коду краща за швидкість роботи програми, простота програми краще за її складність, ясність коду краща за його хитромудрість, а безпека програми краща за її ненадійність. Набагато простіше зробити коректну програму швидшою, ніж швидко програму коректною [2].

Запам'ятайте такі висловлювання: *«Програма має бути написана для людини, яка її читатиме, і тільки попутно — для машини, яка її виконуватиме»* (Harold Abelson). *«Важливість простоти дизайну неможливо переоцінити»* (Jon Bentley).

Масштабованість даних та складність алгоритмів. У процесі експлуатації програми можливе зростання даних. Алгоритми, які працюють з даними, повинні мати передбачуване і, бажано, не гірше, ніж лінійно залежить від кількості оброблюваних даних під час роботи програми. Будь-який сьогоdnішній код завтра може мати справу з великими обсягами даних. Це

означає, що слід уникати вбудовування в програми того, що стане пасткою під час роботи з великими файлами, великими базами даних, з великою кількістю пікселів, великою кількістю вікон, процесів, каналів зв'язку тощо. Слід уникати використання алгоритмів, які працюють з даними користувача, що можуть зростати, але не здатні до масштабування. Для подолання складнощів через масштабованість даних і збільшення складності коду можна скористатися такими рекомендаціями [2]:

- використовуйте гнучкі динамічні дані замість масивів фіксованого розміру;
- слід точно знати складність алгоритму, що використовується, інакше лінійний алгоритм, який викликає іншу лінійну операцію, в результаті робить алгоритм квадратичним;
- віддавайте перевагу алгоритмам з лінійною складністю $O(N)$ або логарифмічною складністю $O(\log N)$;
- ніколи не використовуйте експоненціальний алгоритм, наприклад, пошук найдовшої спільної підпоследовності, де навіть невелике збільшення даних призводить до істотного падіння продуктивності.

Запобігання передчасній оптимізації. *«Передчасна оптимізація – корінь усіх бід»* (Donald Knuth). Передчасна оптимізацію визначається як ускладнення дизайну або коду, що робить його менш легким для читання в ім'я підвищення продуктивності, коли зусилля не виправдовуються. Щоб зробити код швидким, в першу чергу його треба зробити максимально зрозумілим і зручним для читання. Ясний код простіше написати коректно, простіше зрозуміти, простіше переробити і простіше оптимізувати. Ускладнення, включаючи оптимізацію, завжди можна зробити пізніше, і лише за необхідності. Якщо потрібно оптимізувати код, почніть з оптимізації алгоритмів [2].

Мінімізація глобальних даних та тих, що спільно використовуються. Слід уникати даних із зовнішнім зв'язуванням у сфері видимості простору імен. Їхнє застосування ускладнює логіку програми і призводить до тісного зв'язку між різними і віддаленими частинами програми. Дані, що спільно використовуються роблять, менш ефективним тестування процедур (функцій) і модулів, оскільки коректність фрагмента коду, що використовує загальні дані, обумовлена історією зміни цих даних, зумовлює функціонування коду, який використовуватиме ці дані пізніше [2].

Резюме

Процедурно-орієнтоване програмування — це парадигма програмування, що походить від імперативного програмування, заснованого на концепції викликів процедур.

Процедурно-орієнтована програма, по суті, є послідовністю викликів процедур, складні процедури можна декомпозувати на процедури меншої складності за принципом: одна процедура – один логічний блок.

Переваги використання процедурного програмування: простота, ефективність, портативність.

Недоліки використання процедурного програмування: обмежена гнучкість і складність розширення коду; складності супроводу і модифікації великих програм

Переваги структурного програмування: зручність і легкість для розуміння, програми легше читати та вивчати, помилки легше знайти, програми легше обслуговувати, підвищення продуктивності розробки прикладних програм.

Основні правила проєктування функцій: функція повинна розділити структуру програми на окремі відносно незалежні блоки; одна функція має розв'язувати одну локальну задачу; потрібно уникати довгих функцій і глибокої їх вкладеності; функція не повинна використовувати параметри без необхідності; функція `main()` є коротким описом усієї програми, є точкою входу до програми і виконує роль диспетчера, що викликає інші функції програми; потрібно уникати «ланцюгових» викликів функцій.

Розробка процедурної програми ведеться покроково методом «зверху вниз» за принципом «розділяй і володарюй»

Принцип «розділяй і володарюй» означає поділ великої задачі на менші частини, як наслідок просте вирішення малих підзадач, що в свою чергу породжує вирішення великої задачі.

Реалізація принципу «розділяй і володарюй» здійснюється через застосування процедур (функцій).

Принцип простоти означає, що слід уникати непотрібних складнощів під час проєктування та програмування.

Принцип повторного використання коду свідчить про доцільність блоків коду, що повторюються, виносити у процедури (функції) і писати код один раз, а викликати функцію стільки разів, скільки потребує алгоритм програми.

Контрольні запитання

1. Що таке проєктування (design) програмного забезпечення?
2. У чому полягають принципи структурного проєктування?
3. Які є принципи процедурного програмування?
4. Поясніть сутність таких принципів: «розділяй і володарюй», модульність, програмування зверху вниз або спадна розробка.
5. Визначте переваги та недоліки структурного проєктування та процедурного програмування.
6. Яких правил слід дотримуватися в процесі проєктування функцій?
7. У чому полягає технологія розробки процедурно-орієнтованої програми?
8. За якими ознаками можна визначити правильно спроектовану процедурно-орієнтовану програму?

Контрольні завдання

1. Здійснити алгоритмічну декомпозицію та розробити HIPO (Hierarchical Input Process Output) діаграму такої задачі: згенерувати список натуральних чисел, задавши їх кількість; вивести створений список на екран, виконати такі операції: додати нові елементи на задану позицію в списку; підрахувати кількість парних елементів в списку; замінити усі парні значення списку на середнє арифметичне елементів списку.

2. Задані блоки HIPO діаграми: Система управління запасами, Оновити запас, Процес продажу, Процес повернення, Процес відправлення, Створити звіт, Відобразити звіт про стан запасів, Увести інвентарний дані, Змінити запас, Додати запас, Видалити запас.

Розставити ці блоки в HIPO діаграмі відповідно до логіки програми управління запасами на виробництві. Кожний блок пронумерувати за рівнями (1, 2, 3, 4), підрівнями (1.1, 1.2, 2.1, 2.2, ...), блоками 3-го рівня підпорядкованості (1.1.1., 1.1.2, ...).

3. Проаналізуйте код лабораторних робіт з дисципліни «Основи програмування» з точки зору вимог структурного і процедурно-орієнтованого програмування. Відмітьте усі невідповідності вашого коду вимогам C/C++ Coding Standard. Перепишіть код з дотриманням вимог C/C++ Coding Standard. Зафіксуйте свою роботу по рефакторингу коду у звіті до лабораторної роботи.

2 Стиль кодування

2.1 Визначення імен

2.1.1 Аббревіатури (скорочення)

Аббревіатури часто використовуються в програмуванні на C++ для позначення скорочених слів чи фраз. Приклади аббревіатур та їх значень у C++ [5]:

- STL – Standard Template Library: Це колекція шаблонів класів та функцій, які допомагають вирішувати різноманітні завдання, такі як сортування, пошук, та інші операції над даними;
- IDE – Integrated Development Environment: Інтегроване середовище розробки, як, наприклад, Visual Studio або Code::Blocks;
- OOP – Object-Oriented Programming: Об'єктно-орієнтоване програмування – методологія розробки програм, яка базується на об'єктах та класах;

- RAI – Resource Acquisition Is Initialization: Це ідіома програмування, яка вказує на те, що ресурси (наприклад, пам'ять) повинні виділятися під час ініціалізації об'єкта і звільнятися під час його знищення;
- CMake – Cross-platform Make: Це система автоматизації збірки програмного коду на різних платформах;
- API – Application Programming Interface: Це набір функцій та процедур, доступних для використання у програмах;
- SQL – Structured Query Language: Мова запитів, яка використовується для взаємодії з базами даних;
- GUI – Graphical User Interface: Графічний інтерфейс користувача, що надає зручний спосіб взаємодії з програмою через вікна, кнопки та інші графічні елементи;
- UTF-8 – Unicode Transformation Format-8: Система кодування для тексту, яка підтримує широкий спектр символів і мов.

Використовувати аббревіатури необхідно для визначення імен функцій та змінних. Аббревіатури повинні у своїй назві відображати зміст самої сутності.

Гарний стиль програмування

```
int calculatedValue;
float calculateValue(int a, int b int c)
```

Поганий стиль програмування

```
int cV;
float calcVal(int a, int b int c)
```

2.1.2 Формати багатослівних ідентифікаторів

Багатослівні ідентифікатори – це імена змінних, функцій, класів і інших об'єктів, які складаються з кількох слів або скорочень. У мові C++ заборонено використання пробілів як роздільників багатослівних ідентифікаторів. Можна писати назви, просто з'єднуючи слова, але це не є гарним стилем програмування. Правильне форматування полегшує читання та розуміння коду, зменшує можливість помилок і допомагає підтримувати єдність в кодовій базі.

Один із підходів полягає в тому, щоб розділяти окремі слова двома символами [3]: дефіс («-») і підкреслення («_»), останнє стало відоме як snake_case («змійний регістр»):

```
int my_variable_name = 0;
void calculate_total_sum();
class my_class_name;
```

Знаки підкреслення у верхньому регістрі, як у UPPER_CASE, зазвичай використовуються для макросів препроцесора C, тому відомі як MACRO_CASE.

Також існує підхід, що полягає у позначенні межі слів за допомогою середньої літери, що називається «camelCase», «PascalCase»:

```
int myVariableName;
void calculateTotalSum();
class MyClassName;
```

Загальні правила багатослівних ідентифікаторів [3]:

- назви повинні бути досить інформативними, щоб легко зрозуміти їх призначення;
- уникати скорочень, якщо вони не є загальноприйнятими або несуть специфічний сенс;
- дотримуватися однакового стилю форматування в рамках проекту або команди розробників.

2.1.3 Мнемонічність ідентифікаторів

Мнемонічність ідентифікаторів (назв змінних, функцій тощо) є важливою частиною розробки програмного коду, оскільки допомагає полегшити розуміння і підтримку коду колегами, в тому числі вам самим, при подальшій роботі з ним. Ось декілька загальних принципів щодо мнемонічності ідентифікаторів [5]:

- можливість розшифрувати призначення змінної: ідентифікатори мають бути такими, щоб їх легко було розшифрувати, тобто зрозуміти, що робить змінна або функція. Замість x або foo варто використовувати назви, які описують призначення змінної, наприклад, counter або CalculateTotal();
- короткість: назви мають бути досить короткими, але при цьому інформативними. Вони повинні використовувати мінімальну кількість символів, не втрачаючи при цьому чіткості. Наприклад, num для "number" або str для "string";

- **змістовність:** назви повинні містити ключові слова чи терміни, які описують сутність ідентифікатора. Наприклад, якщо у вас є функція для додавання чисел, назва `AddNumbers()` буде більш інформативною, ніж просто `Add()`;
- **унікальність:** у межах одного контексту (наприклад, в межах однієї функції) ідентифікатори мають бути унікальними. Це допомагає уникнути плутанини та конфліктів між змінними чи функціями зі схожими назвами;
- **спільнота:** користуйтеся загальноприйнятими конвенціями та стандартами назв. Наприклад, велика літера в початковому регістрі для назв змінних і малими літерами для функцій у багатьох мовах програмування є стандартним підходом;
- **якість коментарів:** додайте коментарі до складних або незрозумілих ідентифікаторів, щоб пояснити їх призначення і використання;
- **використання англійської мови:** якщо можливо, використовуйте англійську мову для ідентифікаторів, оскільки вона є загальноприйнятою в програмуванні і полегшує роботу в команді.

Мнемонічність ідентифікаторів сприяє покращенню розуміння та підтримки програмного коду, зменшує ймовірність помилок та полегшує співпрацю в командному середовищі [6].

2.1.4 Імена, що визначають тип

Поняття типу даних є одним із основних понять програмування. В C++ типи можна класифікувати за декількома основними назвами [7]:

Фундаментальні типи даних:

- `int`: представляє цілі числа.
- `float`: числа з одинарною точністю з плаваючою комою.
- `double`: числа з подвійною точністю з плаваючою комою.
- `char`: символ.
- `bool`: булеві значення представлені ключовими словами `true` і `false`.
- `void`: вказує на відсутність конкретного значення.

Похідні типи даних:

- масив (Array):** колекція елементів одного типу даних;
- показчик (Pointer):** зберігає адресу пам'яті іншого типу даних;
- посилання (Reference):** псевдонім існуючої змінної.

Користувацькі типи даних:

- `struct`: визначає складний тип даних, який групує пов'язані змінні;
- `class`: реалізує клас, який інкапсулює дані і функції;
- `union`: структуру даних, яка може містити один із кількох типів даних;
- `enum`: перелік з іменованими значеннями.

Модифікатори типів:

- `const`: вказує, що значення змінної не може бути змінено;
- `volatile`: показує, що значення змінної може змінюватися несподівано;
- `mutable`: дозволяє змінювати дані члена класу в `const`-функції.

Псевдоніми типів (Type Aliases):

- `typedef`: Створює псевдонім для існуючого типу даних;
- `using`: Альтернативний спосіб створення псевдонімів типів (стандарт C++20).

Типи стандартної бібліотеки (Standard Library Types):

- `string`: представляє рядок символів;
- `vector`: динамічний масив;
- `map`: реалізує асоціативний масив (пари ключ-значення).

2.1.5 Імена файлів

Імена файлів також повинні відповідати правилам гарного стилю програмування.

Загальні рекомендації щодо імен файлів у C++ [8]:

- Імена файлів повинні бути короткими і описовими.

- Імена файлів повинні бути узгоджені між собою.
- Імена файлів повинні бути узгоджені з іменами класів і функцій, які вони містять.
- Використовуйте великі літери для імен класів, а малі літери для імен функцій і змінних.
- Використовуйте дефіси для розділу імен файлів, що складаються з декількох слів.
- Використовуйте імена файлів, які легко запам'ятати і зрозуміти.
- Використовуйте імена файлів для організації коду.

Поради щодо імен файлів у C++:

- Використовуйте розширення .cpp для файлів з вихідним кодом.
- Використовуйте розширення .h для заголовкових файлів.
- Використовуйте розширення .O для об'єктних файлів.
- Використовуйте розширення .exe для виконуваних файлів.

Прикладів імен файлів на C++:

- MyClass.cpp – файл з вихідним кодом для класу MyClass.
- my_function.cpp – файл з вихідним кодом для функції my_function.
- my_variable.cpp – файл з вихідним кодом для змінної my_variable.
- main.cpp – файл з вихідним кодом для основної програми.

Крім того, можна використовувати імена файлів для організації коду. Наприклад, можна створити каталог для кожного класу або функції. Це може зробити код більш структурованим і легшим для розуміння.

Приклади організації коду за допомогою імен файлів:

classes/MyClass.cpp – файл з вихідним кодом для класу MyClass.

functions/my_function.cpp – файл з вихідним кодом для функції my_function.

variables/my_variable.cpp – файл з вихідним кодом для змінної my_variable.

main.cpp – файл з вихідним кодом для основної програми.

Також дуже важлива деталь – це самодостатність заголовкових файлів. Самодостатній заголовковий файл – це такий заголовковий файл, який містить всі необхідні визначення і прототипи функцій для використання класу чи функції, які він містить. Це означає, що компілятор може скомпілювати програму, використовуючи лише сам заголовковий файл, без необхідності включати будь-які інші заголовкові файли [10].

2.1.6 Імена просторів імен, класів, інтерфейсів, подій, властивостей

Простори імен, класи, інтерфейси, події та властивості є одними з ключових понять мови програмування C++.

Простір імен – це область визначення змінних, типів, констант та функцій, яка об'єднана в єдиний іменований або неіменований блок. Простір імен позначається з використанням ключового слова namespace [11]:

```
namespace MyNamespace
{
    int myVariable = 0;
    void myFunction()
    {
        // Do something
    }
}
```

Клас – це шаблон для створення об'єктів. Клас містить поля (змінні) та методи (функції), які можуть бути викликані для цих об'єктів [12]:

```
class MyClass
{
public:
    int myVariable;
    void myMethod()
    {
        // Do something
    }
};
```

Інтерфейс – це набір методів без їх реалізації. Інтерфейси використовуються для реалізації поліморфізму. Приклад оголошення інтерфейсу [12]:

```
class MyInterface
{
    public:
    virtual void myMethod() = 0;
};
```

Подія – це спосіб повідомлення про те, що сталося в програмі. Події можуть бути згенеровані класами або користувачем [12]:

```
class MyEvent
{
    public:
    void addListener(void (* listener)())
    {
        // Do something
    }
};
```

Властивості – це способи доступу до змінних класу. Властивостей не існує як окремого типу даних в C++, але їх можна реалізувати за допомогою методів-аксесорів (getter) та методів-мутаторів (setter) [12]:

```
class MyClass
{
    private:
    int myVariable;
    public:
    int getMyVariable()
    {
        return myVariable;
    }
    void setMyVariable(int value)
    {
        myVariable = value;
    }
};
```

2.1.7 Визначення імен констант

У C++ константи – це сутності, значення яких не можна змінити. Вони використовуються для зберігання значень, які не повинні змінюватися впродовж виконання програми.

Існують такі правила визначення імен констант [5]:

Константи повинні бути іменовані великими літерами, розділеними підкресленнями. Це дозволяє легко відрізнити їх від змінних, які, як правило, іменуються малими літерами:

```
const int MAX_VALUE = 100;
const double PI = 3.14159;
```

Імена констант повинні бути короткими та описовими. Вони повинні чітко відображати значення, яке вони представляють.

Імена констант повинні бути узгоджені з загальноприйнятими конвенціями. Наприклад, константи, що представляють фізичні величини, повинні іменуватися відповідно до Міжнародної системи одиниць (SI).

Уникати числових значень у назвах констант, замість цього використовуйте докладні та описові імена.

Розміщувати константи у відповідних файлових чи класових областях. Це допомагає уникнути збігів назв констант у різних частинах коду.

Дотримуватися консистентних правил визначення імен констант у всьому проекті, щоб полегшити розуміння та обслуговування коду.

Константи, які представляють імена, повинні закінчуватися словом "_NAME":

```
const char* AUTHOR_NAME = "John Doe";
```

2.1.8 Визначення імен функцій

Розглянемо загальні принципи визначення імен функцій [2]:

1. Використання імен, які зрозумілі навіть людям з іншої команди розробників.
2. Ім'я має говорити про мету чи застосовність функції.
3. Не заощаджуйте на довжині імені, краще довше і зрозуміліше.
4. Уникати аббревіатур.

Приклад хорошого стилю визначення імен функції:

```
calculatePolynomValue(); generateRandomData();
```

Приклад поганого визначення імен функції:

```
calcPv(); grData();
```

2.1.9 Визначення імен параметрів та аргументів функцій

Визначення імен параметрів та аргументів функцій є важливою частиною хорошого стилю програмування, оскільки вони впливають на зрозумілість та читабельність коду. Дотримання правильного стилю при визначенні імен допомагає зробити код більш зрозумілим та легше підтримуваним.

Загальні рекомендації щодо визначення імен параметрів та аргументів функцій [7]:

1. Дескриптивні назви.

Назви параметрів та аргументів повинні бути якнайбільш докладними та описовими. Це допомагає іншим розробникам розуміти, які дані або значення очікується у функції.

Приклад поганого стилю: `void Calculate(int a, int b);`

Приклад хорошого стилю: `void Calculate(int operand1, int operand2);`

2. Специфікація типу.

Додайте до імен параметрів та аргументів інформацію про їхні типи даних. Це полегшує розуміння, які типи даних очікуються функцією.

Приклад поганого стилю: `void PrintValue(x);`

Приклад хорошого стилю: `void PrintValue(int value);`

3. Виключення змішаних регістрів.

Використовуйте лише один регістр (зазвичай *snake_case* або *camelCase*) для імен параметрів та аргументів. Уникайте змішаних великих і малих літер або інших неочікуваних регістрів.

Приклад поганого стилю: `void CalculateValue(int Value);`

Приклад хорошого стилю: `void Calculate_value(int value);`

4. Уникайте скорочень.

Не використовуйте скорочення або аббревіатури для імен параметрів та аргументів, які можуть бути незрозумілими іншим розробникам.

Приклад поганого стилю: `void calc(int val);`

Приклад хорошого стилю: `void Calculate(int value);`

5. Уникайте конфліктів імен.

Переконайтеся, що ім'я параметра функції не конфліктує з іменем змінної в більш високому рівні області видимості.

Приклад поганого стилю:

```
int x = 10; void Calculate(int x)      // Конфлікт імен
```

Приклад хорошого стилю:

```
int x = 10; void Calculate(int value)  // Імена не конфліктують
```

6. Використання слів з великої літери для багатослівних імен.

Якщо ім'я параметра складається з більш ніж одного слова, використовуйте верблужий стиль для розділення слів, зробіть перше слово з великої літери для імен функцій, зробіть перше слово з маленької літери для параметрів функцій.

Приклад поганого стилю: `void calculateValueOfVariable(int valueofvariable);`

Приклад хорошого стилю: `void CalculateValueOfVariable(int valueOfVariable);`

Зберігайте послідовність та стиль імен такими, що роблять ваш код більш зрозумілим і легким для сприйняття іншими розробниками. Це полегшить роботу вам та вашій команді при розробці та підтримці програмного забезпечення.

2.1.10 Визначення імен типів шаблонів

В C++ визначення імен типів шаблонів (template types) визначається загалом тими ж правилами, що і визначення імен інших типів даних [8]:

1. Дотримуйтесь стилів визначення імен. Важливо дотримуватися стилю визначення імен, який використовується в вашому проєкті або команді. Ось кілька загальних стилів:

Camel Case: імена мають вигляд "camelCase". Починаючи з малої літери, кожне нове слово розпочинається з великої літери без пробілів. Наприклад, myTemplateType.

Snake Case: імена мають вигляд "snake_case". Усі літери малі, а слова розділяються нижніми підкресленнями. Наприклад, my_template_type.

Pascal Case (CamelCase з великої початкової літери): імена мають вигляд "PascalCase". Починаючи з великої літери, кожне нове слово розпочинається з великої літери без пробілів. Наприклад, MyTemplateType.

2. Описові назви: ім'я типу шаблону повинно бути досить описовим, щоб легко розуміти його при читанні коду.

3. Уникайте загальних імен: спробуйте уникати використання занадто загальних імен, таких як T або Type. Краще використовувати імена, які більше відображають призначення типу.

4. Зазвичай ім'я класу шаблону повинно відрізнятися від імені змінної. Наприклад:
template < typename T >

```
class MyTemplateClass
{
    // ...
};

int main()
{
    MyTemplateClass<int> myInstance;
    // ...
}
```

5. Узгодженість: важливо, щоб імена ваших типів шаблонів були узгоджені з іншими частинами вашого коду і відповідали загальному стилю.

Загалом, дотримання правил визначення імен та стилів допомагає покращити читабельність та розуміння вашого коду і спрощує спільну роботу з іншими програмістами.

2.1.11 Визначення імен ітераторів (i, j, k, l, m, n)

Лічильник циклу — це цілочисельна змінна, яка оголошується з єдиною метою: рахувати, скільки разів виконався цикл.

Лічильникам циклу часто дають прості імена, такі як i, j чи k. Проте в цих іменах є одна серйозна проблема. Якщо ви хочете дізнатися, де у вашій програмі використовується лічильник циклу і скористаєтеся функцією пошуку символів i, j чи k, то в результаті отримаєте половину своєї програми, так як i, j та k використовуються в багатьох іменах. Отже, краще використовувати iii, jjj чи kkk в якості імен для лічильників. Вони більш унікальні, їх значно простіше знайти, і вони виділяються в коді. А ще краще використовувати «реальні» імена для змінних, наприклад, count або будь-яке інше ім'я, яке надає контекст використання цієї змінної [6].

2.1.12 Визначення імен логічних змінних та функцій (sFound, isOpen)

Правила визначення імен логічних змінних та функцій у C++ досить стандартні і базуються на загальноприйнятих звичаях для стилів програмування.

Для визначення імен змінних [9]:

- використовуйте імена, які описують суть змінної. Наприклад, isReady замість flag1, hasValue замість bool1;
- використовуйте camelCase або snake_case для визначення імен змінних. Наприклад, isReadyToGo або has_value;

- спробуйте уникати коротких абревіацій. Якщо ви використовуєте абревіатуру, вона повинна бути зрозумілою і широко прийнятою;
- вибирайте описові імена, які допоможуть іншим розуміти, для чого призначена змінна.

Приклади імен змінних:

```
isAvailable
userCount
totalAmount
maxValue
```

Для визначення імен функцій використовуйте дієслова або дієслівно-іменникові фрази для опису функцій. Описуйте функціональність функції у її імені так, щоб інші розробники зрозуміли, що робить ця функція. Наприклад, `isValidInput` або `canFileBeOpen`.

2.1.13 Визначення імен перелічувань

Для визначення імен перелічувань в C++ зазвичай використовують CamelCase нотацію [6]. Це означає, що кожне слово в імені перелічування починається з великої літери, окрім першого слова. Наприклад, якщо ми створюємо перелічування для днів тижня, ми можемо назвати його `DaysOfWeek`.

Якщо перелічування має багато значень, то краще використовувати `snake_case` нотацію [6]. В цьому випадку ім'я складається з малих літер, а слова розділяються символом підкреслення. Наприклад, якщо ми створюємо перелічування для кольорів райдуги, ми можемо назвати його `rainbow_colors`.

Якщо вам потрібно створити перелічування з одним значенням, то можна використовувати будь-яку з нотацій. Наприклад, якщо ми створюємо перелічування для напрямку руху, ми можемо назвати його `Direction` або `direction`.

Приклади оголошення перелічень:

<pre>enum class DaysOfWeek { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };</pre>	<pre>enum class rainbow_colors { Red, Orange, Yellow, Green, Blue, Indigo, Violet };</pre>	<pre>enum class Direction { North, South, East, West };</pre>
--	--	---

За замовчуванням значення перелічень починаються з 0, але початкове значення та значення елементів перелічення можливо змінити. Приклади оголошення перелічень зі зміненими значеннями

```
enum Result
{
    SUCCESS = 0,
    ERROR = -1,
    INVALID = -2
};
```

```
enum Months
{
    JAN = 1,
    FEB,
    MAR,
    APR,
    MAY,
    JUN,
    JUL,
    AUG,
    SEP,
    OCT,
    NOV,
    DEC
};
```

2.1 Використання змінних і констант (літералів)

2.2.1 Локалізація змінних

«Гарним стилем» програмування є визначення кожної змінної якомога локальніше, тобто коли ви володієте достатньою інформацією для ініціалізації і безпосередньо перед її першим використанням [9]. Ніколи не оголошуйте змінну до того, як ви зможете її належним чином ініціалізувати. Часто трапляється таке що ви не маєте достатньо інформації для ініціалізації змінних. Це залишає два варіанти: або ініціалізувати пустим значенням за замовчуванням, що зазвичай може призвести до помилок, якщо змінну використати до того, як надати їй правильне значення, або залишити її неініціалізованою, що є небезпечним. Неініціалізована змінна користувацького типу самоініціалізується до деякого порожнього значення.

```
//Поганий стиль програмування
23 int objectsCount;
...
568 objectsCount = calculateObjectsCount();
569 if (objectsCount > 0)
//Гарний стиль програмування
568 int objectsCount = calculateObjectsCount();
569 if (objectsCount > 0)
```

2.2.2 Ініціалізація змінних при їх оголошенні

Ініціалізація змінних при їх оголошенні означає, що змінні потрібно оголошувати як можна локальніше, тобто в самому вузькому контексті, в якому вони необхідні. Замість того щоб оголосити змінну в початку функції, її можна оголосити в конкретному блоці, який її використовує.

Правила ініціалізації змінних [8]:

1. Змінні, оголошені в більш широкому контексті мають більш тривалий життєвий цикл, це може призвести до того, що вони будуть використані в місцях, де вони не потрібні. Це може ускладнити розуміння та обслуговування цього коду.

2. Також такі змінні можуть бути видимими з інших частин коду, що може призвести до конфліктів імен змінних.

3. Іноді теж може трапитись помилка, коли змінна ініціалізована в широкому контексті та до змінної звертаються до її ініціалізації.

Проте, існують виключення з правил [10]:

1. Іноколи може бути корисно оголосити змінну на початку функції, щоб її можна було використовувати в декількох місцях, наприклад, масив, з яким буде проводитись робота не один раз.

2. Константи не мають життєвого циклу, тому до них це правило не відноситься.

2.2.3 Використання макросів

Макроси – найгрубіший інструмент для абстракції в C/C++, вони не є функціями, займають багато місця в таблицях символів, вони застосовуються під час попередньої обробки, перед тим, як будь-які синтаксичні і семантичні правила C++ почнуть діяти. Необхідно уникати використання макросів [2].

Приклад макросів:

```
#define HELLO printf("Hello World! \n") //оголошення макросів
#define FOR for(int i=0; i<4; i++)

int main(void)
{
    FOR HELLO; //4 рази виведе на екран Hello World!
    return 0;
}
```

2.2.4 Використання «магічних чисел»

Магічне число або сигнатура – цілочисельна або текстова константа, яка використовується для однозначної ідентифікації ресурсу або даних. Також «магічними числами» називають погану практику програмування, коли у тексті програми зустрічається числове значення, сенс якого не очевидний. Для уникнення використання «магічних чисел» варто замінити «магічні числа» на константи, та зберігати їх окремо від коду (блок констант, окремий ресурсний файл) [3].

Приклади:

1. Використання «магічного числа» 73:

```
for(int i = 1; i < 73; i++)
{
    // Do something
}
```

2. Заміна використання «магічного числа» 73:

```
const int LOOP_ITERATIONS = 73;
...
for(int i = 1; i < LOOP_ITERATIONS; i++)
{
    // Do something
}
```

3. Важливі константи, специфічні для області на рівні простору імен.

```
const size_t PAGE_SIZE = 8192,
WORDS_PER_PAGE = PAGE_SIZE / sizeof(int),
INFO_BITS_PER_PAGE = 32 * CHAR_BIT;
```

4. Константи специфічні для класу

```
// File widget.h
class Widget
{
    static const int defaultWidth = 400;
    static const double defaultPercent;
    static const char* Name() { return "Widget"; }
};
// File widget.cpp
const double Widget::defaultPercent = 66.67;
const int Widget::defaultWidth;
```

2.2.5 Оголошення типів

В C++ правила оголошення типів включають наступні аспекти [5]:

- при оголошенні змінних вказуйте їхні типи перед іменами змінних:

```
int age;
double price;
```
- оголошення констант: якщо змінну плануєте використовувати як константу, використовуйте ключове слово `const`:

```
const int numberOfMonths = 12;
```
- типи мають бути інформативними: назви типів і змінних мають бути інформативними та описувати їхню природу і призначення.
- для оголошення складних типів даних чи аліасів (псевдонімів) типів, можна використовувати `typedef` (C++20) або `using`:

```
typedef double Price;
using Score = int;
```
- при оголошенні функцій слід вказувати їхні типи повернення, імена та список параметрів:

```
int add(int a, int b);
double calculateAverage(double values[], int count);
```
- при оголошенні класів та структур слід вказувати доступ до членів (`public`, `private`, `protected`).

```
class Student
{
    public:
        // ...
    private:
        // ...
};
```

- правила визначення імен: користуйтеся правилами визначення імен для типів, де імена починаються з великої літери (CamelCase) або малими літерами з підкресленнями (snake_case).

Використання ключових слів `class` та `struct`. В C++ різниці між ключовими словами `class` і `struct` полягають лише у рівні доступу за замовчуванням (`private` для `class` і `public` для `struct`). Вибирайте той, який відповідає вашим потребам.

Оголошення або ініціалізація. Зазвичай оголошення змінних відділяється від їхньої ініціалізації. Оголошення виконується ключовим словом `extern`, а ініціалізація - прирівнюванням значення:

```
extern int x; // Оголошення
x = 10;      // Ініціалізація
```

Використання просторів імен. Щоб уникнути конфліктів імен, використовуйте простори імен:

```
namespace MyNamespace
{
    int value;
}
```

Ці правила допомагають зробити код більш зрозумілим, зменшити можливість помилок і полегшити співпрацю в команді.

2.2.6 Оголошення покажчиків і посилань в C/C++ та їх ініціалізація

Покажчик – це змінна, яка містить адресу місця в пам'яті. Посилання – це простий тип даних, який менш потужний, але більш безпечний ніж вказівник. Тип посилання іноді називається аліасом, і слугує для створення об'єкту додаткового імені. Посилання дозволяє маніпулювати об'єктом так само, як це робиться за допомогою вказівника, однак це не потребує спеціального синтаксису, яке необхідне при роботі з вказівниками. Зазвичай посилання використовуються як формальні параметри функцій [10].

Правила використання покажчиків та посилань [10]:

1. Завжди ініціалізуйте ваші покажчики і посилання. Це допомагає уникнути неконтрольованих значень і зменшити ризик помилок.

```
int* ptr = nullptr; // або int* ptr = 0;
int& ref = someValue;
```

2. Використовуйте правильні типи покажчиків і посилань: Вибирайте відповідний тип покажчика або посилання в залежності від завдання. Наприклад, використовуйте `const` покажчики, де це можливо, для запобігання незмінності даних.

```
const int* readOnlyPtr = &value;
```

3. Використовуйте сучасний синтаксис C++20.

4. Дотримуйтесь чіткої структури: Розміщуйте покажчики і посилання на тих рядках, де вони знадобляться, і намагайтесь уникати скупчення покажчиків в одному рядку.

```
int* ptr1 = nullptr;
int* ptr2 = nullptr;
```

5. Використовуйте ключові слова `new` і `delete` обережно, будьте особливо уважні, щоб уникнути витоку пам'яті.

```
int* dynamicPtr = new int;
delete dynamicPtr;
```

2.2.7 Літерали

Літерали (або літеральні константи) – це значення, які використовуються безпосередньо в коді.

2.2.7.1 Цілочисельні типи

У C++ існує 4 цілочисельні типи: short, int, long, long long (табл. 1). Діапазон цілочисельної змінної визначається двома факторами: її розміром (в бітах) і її знаком (який може бути signed або unsigned) [9].

Таблиця 1. – Діапазони значень цілочисельних типів C++

Розмір/Тип	Діапазон значень
1 байт signed short	від -128 до 127
1 байт unsigned short	від 0 до 255
2 байти signed int	від -32 768 до 32 767
2 байти unsigned int	від 0 до 65 535
4 байти signed long	від -2 147 483 648 до 2 147 483 647
4 байти unsigned long	від 0 до 4 294 967 295
8 байт signed long long	від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
8 байт unsigned long long	від 0 до 18 446 744 073 709 551 615

Гарний стиль використання літералів цілочисельних типів:

1. Використовуйте цілочисельні типи signed, замість unsigned.
2. Використовуйте той тип даних, що відповідає потребам.

```
long long count_of_lessons = 4;           // погано, зайві витрати пам'яті
short count_of_lessons = 4;               // добре
```

3. Ніколи не допускайте можливості виникнення переповнення в ваших програмах.

2.2.7.2 Дійсні типи

Літерали дійсних типів представляють собою константи з числами з плаваючою точкою.

У C/C++ є декілька видів літералів дійсних типів [5]:

1. Літерали з плаваючою точкою:

- десяткові літерали:

```
double pi = 3.14159265;
```

- експоненціальні літерали (представлення числа з використанням «e»):

```
double expnum = 1.23e-4; // Має значення 1,23 * 10^-4
```

- літерали у шістнадцятковому форматі (представляють число в шістнадцятковому форматі з префіксом «0x»):

```
double hex = 0x1A.5; // 26.3125 в десятковому
```

2. Літерали типу float.

Літерал типу float є менший за розміром, ніж double. Щоб вказати літерал типу float, ви можете додати суфікс «f» або «F» до значення. За замовчуванням, літерали дійсних чисел без суфіксу вважаються типом double.

```
float pi = 3.14f;
```

3. Літерали типу long double.

Літерал типу long double є більший за розміром, ніж double. Щоб вказати літерал типу long double, ви можете додати суфікс «L» або «l» до значення.

```
long double pi = 3.14159265358979323846L;
```

2.2.7.3 Символьний тип

Символьний літерал складається із символної константи. Він представлений символом, оточеним одними лапками. Існує п'ять типів символних літералу [9]:

- звичайні символні літерали типу char: 'a';
- символні літерали UTF-8 типу char: u8'a';
- розширені символні літерали типу wchar_t: L'a';
- символні літерали типу char16_t UTF-16: u'a';
- символні літерали типу char32_t UTF-32: U'a';

Символ, який використовується для літералу символів, може бути будь-яким символом, за винятком зарезервованих символів зворотної косої риски (\), однієї лапки (') або нового рядка. Зарезервовані символи можна вказувати за допомогою escape-послідовності. Символи можна

вказувати за допомогою універсальних імен символів, за умови, що тип є досить великим для розміщення символу.

2.2.7.4 Рядковий тип

Рядковий літерал представляє послідовність символів, які разом утворюють рядок із завершальним нулем. Символи мають бути укладені у подвійні лапки. Існують такі типи рядкових літералів [6]:

Вузкий рядковий літерал – це не префіксований, розділений подвійними лапками масив `const char [n]` типу, в якому `n` – довжина масиву в байтах. Звичайний рядковий літерал може містити будь-які графічні символи, крім подвійних лапок (`"`), зворотної косої риси (`\`) або символу нового рядка.

Рядок у кодуванні UTF-8 – це префіксований, розділений подвійним лапками масив типу, що завершується значенням `NULL`, де `n` – довжина закодованого масиву `const char [n]` в байтах. Рядковий літерал з префіксом `u8` може містити будь-які графічні символи, за винятком подвійних лапок (`"`), зворотної косої риси (`\`) або символу нового рядка.

Широкий рядковий літерал – це масив констант `wchar_t`, що має префікс `null`, який має префікс `L` і містить будь-який графічний символ, крім подвійної лапки (`"`), зворотної косої риси (`\`) або символу нового рядка.

Необроблений рядковий літерал – це масив, що завершує значення `NULL`, будь-який тип символу, який містить будь-який графічний символ, включаючи подвійну лапку (`"`), зворотну косу рису (`\`) або новий символ. Необроблені рядкові літерали часто застосовуються в регулярних виразах, які використовують класи символів, а також у рядках HTML та XML.

2.2.7.5 Літерали масиву

У мові програмування C++ літерали масивів (рядків) представляються як послідовності символів, розташованих між двома подвійними лапками. Використовуються для ініціалізації рядків або масивів символів. Ось деякі правила використання літералів масивів (рядків) у C++ [7]:

- подвійні лапки: літерал масиву (рядка) завжди має бути заключений у подвійні лапки. Наприклад, `"hello, world!"` – це правильно, а `'hello, world!'` – неправильно;
- завершення нульовим символом: у C++ рядки завершуються нульовим символом (`'\0'`), що вказує кінець рядка. Під час визначення літерала рядка цей нульовий символ додається автоматично. Наприклад, `"hello"` фактично представляється як масив символів `{ 'h', 'e', 'l', 'l', 'o', '\0' }`;
- символний масив: результатом використання літерала масиву є значення типу `const char[]`, тобто константний масив символів.
- подвійні лапки та одиночні лапки: подвійні лапки слід використовувати для літералів масивів (рядків), одиночні лапки використовуються для символних літералів. Якщо ви помиляєтесь і використовуєте одиночні лапки для рядка, це призведе до помилки компіляції.
- екранування спеціальними символами: якщо ваш рядок містить спеціальні символи, такі як подвійні лапки, зворотні слеші тощо, ви повинні їх екранувати зворотними слешами. Наприклад, `"This is a \"quoted\" word"`.
- Перенос рядка: щоб створити рядок, який розтягується на декілька рядків, ви можете використовувати літерали масивів (рядків) з зворотнім слешем:
`"This is a very long string that \
spans multiple lines."`
- Конкатенація рядків: Ви можете об'єднувати літерали масивів (рядків) разом за допомогою оператора `+`:
`"Hello " + "World!"`

2.2 Вирази

2.3.1 Використання круглих дужок

У мові програмування C++ круглі дужки () використовуються для кількох різних цілей. Основні випадки використання круглих дужок у C++ включають 3:

1. Декларація функцій: круглі дужки використовуються для декларації функцій:
`int add ( int a, int b );`
2. Виклик функцій: круглі дужки використовуються для виклику функцій і передачі аргументів:
`int sum = add ( 3, 5 );`
3. Оператори: круглі дужки використовуються для зміни порядку виразів (пріоритет операцій):
`int result = ( 2 + 3 ) * 4;`
4. Конструктори: круглі дужки використовуються для виклику конструкторів класів.

Наприклад:

```
class MyClass
{
```

```
    public:
        MyClass(int value)
        {
            // конструктор
        }
};
```

```
MyClass obj ( 10 ) ; // виклик конструктора через круглі дужки
```

5. Приведення типів (casting): круглі дужки можна використовувати для приведення типів:

```
double num = 3.14;
int integerNum = static_cast<int>(num);
```

6. Вирази умови (ternary expressions): круглі дужки можна використовувати в тернарних виразах:

```
int max = ( a > b ) ? a : b;
```

7. Для передачі виразів у функції: круглі дужки можна використовувати для передачі виразів у функції:

```
int result = calculate ( 2 + 3 );
```

Гарний стиль:

1. Пропуск між дужками і виразом: зазвичай, якщо ви використовуєте дужки для виразів, додайте пропуск між дужками та виразами для полегшення читання коду:

Гарний стиль програмування	Поганий стиль програмування
<pre>if (x > 0) { // робимо щось }</pre>	<pre>if(x>0) { // робимо щось }</pre>

2. Обгортання умов і виразів: дужки можна використовувати для обгортання складних умов чи виразів, щоб уникнути непорозумінь:

Гарний стиль програмування	Поганий стиль програмування
<pre>if ((x > 0 && y > 0) z < 0) { // робимо щось }</pre>	<pre>if (x > 0 && y > 0 z < 0) { // робимо щось }</pre>

3. Використання у виразах: дужки можна використовувати для виразів для зміни порядку виконання операцій:

Гарний стиль програмування	Поганий стиль програмування
<pre>int result = (a + b) * (c - d) ;</pre>	<pre>int result = a + b * c - d;</pre>

2.3.2 Порівняння логічних змінних із їх значенням

У мові програмування C++, вирази, які перевіряють логічні зміни, спрощено можна порівняти з їх значенням разом в умовних конструкціях.

У нашому прикладі `bool flag; if (flag)` умовний оператор `if` перевіряє значення логічної змінної `flag`. Якщо `flag` дорівнює `true`, то умова визнана істинною і відбувається відповідний блок коду, що слідує за `if`. Якщо `flag` дорівнює `false`, то умова є хибною і відповідний блок коду буде пропущений. Наприклад:

```
bool flag = true;
if (flag)
{
    // Цей блок коду виконається, оскільки flag дорівнює true
}
else
{
    // Цей блок коду не виконається
}
```

2.3.3 Порівняння (==, not =)

У мові програмування C++ оператор порівняння використовується для порівняння значень двох виразів. Для порівняння використовують два основних оператори порівняння: `==` (рівно) та `!=` (не рівно) [9].

Оператор `==` (рівно) використовується для порівняння, чи рівні значень двох виразів. Якщо значення виразів однакові, то вираз з оператором `==` повертає `true`, інакше він повертає `false`.

```
int a = 5;
int b = 5;
if (a == b)
{
    // Цей блок коду виконається, оскільки a є рівним b (5 == 5)
}
```

Оператор `!=` (не рівно) використовується для порівняння, чи не рівні значення двох виразів. Якщо значення виразів неоднакові, то вираз з оператором `!=` повертає `true`, інакше він повертає `false`.

```
int a = 5;
int b = 10;
if (a != b)
{
    // Цей блок коду виконається, оскільки a є не рівним b (5 != 10)
}
```

2.3 Функції та оператори

2.4.1 Передача параметрів за значенням, (інтелектуальному) покажчику або посиланням

Правильний вибір способу передачі аргументів у функцію – за значенням, посиланням або з використанням покажчиків – дуже необхідна навичка, яка допоможе вам зробити ваш код максимально безпечним та ефективним.

Хоча ефективність не повинна бути нашою головною метою, за інших рівних умов, включаючи зрозумілість коду, ми не повинні без потреби знижувати його ефективність.

Вам слід дотримуватись наведених нижче рекомендацій при виборі способу передачі параметрів. Ось рекомендації для вхідних параметрів (які передають інформацію на функцію, але не повертають її) [2]:

- завжди описуйте всі вказівники або посилання на вхідні параметри, як `const`;
- віддавайте перевагу передачі примітивних типів (наприклад, `char` або `float`) і об'єктів-значень з недорогими операціями копіювання (наприклад, `Point` або `Complex<float>`) за значенням;

- вхідні аргументи інших типів користувача краще передавати як посилання на `const`;
- подумайте про передачу за значенням замість передачі за посиланням, якщо функція вимагає створення копії аргументу. Концептуально передача за значенням еквівалентна передачі посилання на константний об'єкт та виконання копіювання, і може допомогти компілятору у виконанні оптимізації усунення тимчасових змінних.

Далі наведено рекомендації для вихідних параметрів (а також для одночасного введення та виведення інформації):

- якщо аргумент необов'язковий, краще передати його як (інтелектуальний) покажчик (що дозволить функції, що викликає, передати нульове значення як вказівку, що аргумент відсутній); та ж рада застосовується і у випадку, якщо функція зберігає копію покажчика або якимось іншим чином працює з належністю аргументу;
- якщо аргумент є обов'язковим і функція не зберігає вказівник на нього або якимось іншим чином впливає на його належність, то такий аргумент краще передавати за посиланням. Це показує, що наявність даного аргументу обов'язково, і змушує функцію, що викликає, відповідати за надання коректного об'єкта.

Не використовуйте функції зі змінною кількістю аргументів.

2.4.2 Код функції не залежить від порядку обчислення її аргументів

Для підвищення швидкодії генерації коду, автори мови C додали розподільвачу регістрів додаткову свободу: при виклику функції порядок обчислення її аргументів залишався невизначеним. Ця мотивація, можливо, стала менш актуальною у сучасних процесорах, але факт залишається фактом: порядок обчислення аргументів не фіксується в C++, і, як виявилось, може значно відрізнятись в різних компіляторах [3].

Це може створити значні проблеми для недбалих програмістів. Розглянемо такий приклад:

```
void Transmogrify( int, int );
int count = 5;
Transmogrify( ++count, ++count ); // порядок обчислення невідомий
```

Все, що можна сказати з впевненістю, це те, що результатом буде число 7, коли виконається тіло функції `Transmogrify`, але нам невідомо, який із аргументів буде 6, а який 7. Ця неоднозначність також поширюється на більш складні випадки, такі як функції, які змінюють свої аргументи як побічний ефект:

```
int Bump( int& x) { return ++x; }
Transmogrify( Bump(count), Bump(count) ); // ще невідомо
```

Зазвичай рекомендується уникати глобальних та спільних змінних. Проте, навіть якщо їх уникати, інші частини коду можуть не дотримуватися цього правила. Наприклад, деякі стандартні функції мають побічні ефекти (наприклад, `strtok` та різні перевантаження `operator<<`, які використовують `ostream`).

Щоб уникнути цих проблем, рекомендується використовувати іменовані змінні для фіксації порядку обчислення. Наприклад:

```
int bumped = ++count;
Transmogrify(bumped, ++count); // в порядку
```

Цим способом можна забезпечити більшу впевненість у порядку обчислення аргументів.

Отже, деяких мовах програмування, зокрема в мові C та C++, порядок обчислення аргументів при виклику функції може залишатися невизначеним. Тому можна сказати, що код функції не залежить від порядку обчислення її аргументів. Також слід підсумувати, що невизначеність може призвести до неочікуваних результатів та невизначеності в коді.

2.4.3 Застосування постфіксної та префіксної форм операторів ++, --

Особливість операторів інкременту і декременту полягає в тому, що у них є префіксна і постфіксна форми з семантикою, що трохи відрізняється. Визначаєте оператори `operator++` і `operator--` так, щоб вони наслідували поведінку своїх вбудованих двійників. Якщо вам не потрібне вихідне значення, використовуйте префіксні версії операторів.

Що стосується `++` і, постфіксні форми операторів повертають вихідне значення, тоді як префіксні форми повертають нове значення. Найкраще реалізовувати постфіксний оператор із використанням префіксного. Ось канонічний вид такого використання [8]:

```
// ---- Префіксні оператори -----
T & T::operator ++()
{ // Префіксний вигляд:
  // Виконання // - Виконання
  // Інкrementа // дій
  return *this; // - Return *this;
}
T & T::operator --()
{ // Префіксний вигляд:
  // Виконання // - Виконання
  // Декременту // дій
  return *this; // - Return *this;
}
//---- Постфіксні оператори -----
T T::operator++(int)
{ // Постфіксний вигляд:
  T old(*this); // - Запам'ятовуємо старе значення
  ++*this;      // - Виклик префіксної версії
  return old;   // - Повернення старого значення
}
T T::operator--(int)
{ // Постфіксний вигляд:
  T old(*this); // - Запам'ятовуємо старе значення
  --*this;      // - Виклик префіксної версії
  return old;   // - Повернення старого значення
}
```

У коді, що викликає, краще використовувати префіксні оператори, якщо тільки вам не потрібне вихідне значення, що повертається постфіксною версією. Префіксна форма семантично еквівалентна, вона вводиться практично так само, і часто трохи ефективніша, оскільки створює на один об'єкт менше.

2.4.4 Використання тернарного умовного оператора ?:

Тернарний умовний оператор є скороченою формою умовного оператора if-else. Він приймає три операнди [5]:

- умова – логічний вираз, який повертає значення true або false;
- значення, якщо умова true – значення, яке буде повернене, якщо умова повертає значення true.
- значення, якщо умова false – значення, яке буде повернене, якщо умова повертає значення false.

Синтаксис:

умова ? значення_true : значення_false

```
int max = x > y ? x : y;
```

Переваги:

- тернарний умовний оператор дозволяє скоротити код;
- він простіший для читання.

Недоліки:

- тернарний умовний оператор може бути складним для розуміння, якщо він використовується в складних логічних виразах.

2.4.5 Оператор if-else, вкладені if

Оператор if-else є умовним оператором, що має такий синтаксис [6]:

```
if (a > 5)
{
  // Виконуємо дії якщо a більше 5
}
else
{
  // Виконуємо дії якщо a менше або дорівнює 5
}
```

Блок `else` є необов'язковим. Також даний блок може містити умову:

```
if (a > 5)
{
    // Виконуємо дії якщо a більше 5
}
else if (a == 3)
{
    // Виконуємо дії якщо a дорівнює 3
}
```

Оператор `if-else` може містити в собі вкладені оператори `if-else`.

```
if ((a > 5) && (a < 8))
{
    cout << " Виконуємо дії якщо a менше 8 і a більше 5" << endl;
}
else
{
    if (a == 5)
    {
        cout << "Виконуємо дії якщо a дорівнює 5" << endl;
    }
    else if (a == 23)
    {
        cout << " Виконуємо дії якщо a дорівнює 23" << endl;
    }
    else if (a == 3)
    {
        cout << " Виконуємо дії якщо a дорівнює 3" << endl;
    }
}
```

Рекомендується вибрати один стиль розміщення дужок і дотримуватися його. Детальніше про правила «гарного стилю» використання описано у розділі 3.3 Форматування.

2.4.6 Оператор `for`

Оператор `for` є одним із операторів циклу у мові програмування C/C++ та має синтаксис [8]:

```
for (ініціалізація; умова; крок)
{
    // Тіло циклу
}
```

У даному циклі можлива відсутність кожного з блоків. Використання циклу `for` рекомендовано коли заздалегідь відома кількість ітерацій (повторень) циклу.

Детальніше про правила «гарного стилю» використання описано у розділі 3.3 Форматування.

2.4.7 Оператор `while`

Цикл `while` є найпростішим з 4 циклів в мові C++. Цикл `while` оголошується з використанням ключового слова `while`. Спочатку оброблюється умова. Якщо її значенням є `true` тоді виконується тіло циклу [10].

```
while (умова повторення циклу)
{
    // Тіло циклу
}
```

Необхідно уникати нескінченних циклів. Вихід з таких циклів слід робити через примусове їх переривання, що порушує вимоги до структурного програмування. Детальніше про правила «гарного стилю» використання описано у розділі 3.3 Форматування.

2.4.8 Оператор `do – while`

Оператор `do– while` є оператором циклу. Тіло циклу `do–while` завжди виконується хоча б один раз. Після виконання тіла циклу перевіряється умова. Якщо вона істинна, то виконання переходить до початку блока `do` і тіло циклу виконується знову [9].

```

do
{
    // Тіло циклу
}
while (умова повторення циклу);

```

Приклад

```

int choice;
do
{
    std::cout << "Please make a selection: \n";
    std::cout << "1) Addition\n";
    std::cout << "2) Subtraction\n";
    std::cout << "3) Multiplication\n";
    std::cout << "4) Division\n";
    std::cin >> choice;
} while (choice != 1 && choice != 2 && choice != 3 && choice != 4);

```

Змінна `choice` повинна бути оголошена поза блоками `do-while`. Якби змінна `choice` була оголошена всередині блоку `do`, то вона була б знищена при завершенні цього блоку ще до перевірки умови `while`.

Загалом, використовувати `do-while` замість `while`, коли потрібно, щоб цикл виконався хоча б один раз, є хорошою практикою.

Гарний стиль форматування `do-while` циклу представлений у розділі 3.3 Форматування.

2.4.9 Оператор switch

Оператор `switch` – є оператором вибору. Поганим стилем програмування є використання вкладених операторів `if-else`, якщо кількість вкладень більше 3. У таких випадках гарним стилем програмування є використання оператора `switch` [5].

```

switch (p)
{
    case '0': // якщо p = 0
    case '1': // якщо p = 1
    case '2': // якщо p = 2
        return true; // повертаємо true якщо p дорівнює 1 або 2 або 3
    default: // інакше повертаємо false
        return false;
}

```

Правила «гарного стилю» використання оператора `switch`:

- використання фігурних дужок для відокремлення початку та кінця оператора;
- використання оператора `break` для виходу з `switch` після виконання умови;
- наявність гілки `default`;
- уникати виконання багатьох операцій в середині `case`;

Гарний стиль форматування оператора `switch` представлений у розділі 3.3 Форматування.

Резюме

Код можна писати по різному. Але, є загальноприйняті правила "гарного стилю" програмування. Дані правила охоплюють використання аббревіатур, визначення імен структурних одиниць коду, використання дужок та операторів, тощо. Дотримання даних правил підвищує ефективність сприйняття коду розробниками завдяки стандартизації написання коду, що підвищує ефективність розробки програмного забезпечення як у команді, так і одному розробнику. також слід пам'ятати про використання пам'яті при написанні програм. Це стосується використання локалізації змінних та використання "правильних" типів даних для збереження даних. У невеликих програмах це непомітно, проте у масштабних додатках це сильно впливає на продуктивність роботи програмного забезпечення. Знання та використання правил "гарного коду" підвищує рівень розробника програмного забезпечення.

Контрольні запитання

1. Що таке аббревіатура? Наведіть приклад поширених аббревіатур. Які правила формування аббревіатур?
2. Що таке багатослівний ідентифікатор? Правила багатослівних ідентифікаторів.
3. Які є принципи мнемонічності ідентифікаторів?
4. Які є типи даних у мові C++?
5. Назвіть правила визначення імен файлів у C++.
6. Які є правила визначення імен просторів імен, класів, інтерфейсів, подій та властивостей?
7. Що таке константа? Правила визначення імен констант.
8. Правила визначення імен функцій.
9. Правила визначення імен параметрів та аргументів функцій.
10. Які є правила визначення імен типів шаблонів?
11. Що таке ітератор? Визначення імен ітераторів.
12. Змінні логічного типу та привила їх визначення імен.
13. Що таке перелічування? Правила визначення імен перелічувань.
14. Правило локалізації змінних.
15. Які є правила ініціалізації змінних?
16. Що таке макрос?
17. Що таке магічне число? Чи можна їх використовувати?
18. Правила оголошення типів.
19. Що таке покажчик? Правила оголошення покажчиків.
20. Що таке літерал? Які є літерали?
21. Літерали дійсних типів.
22. Літерали символьних та рядкового типів.
23. Літерали масиву.
24. Правила використання круглих дужок.
25. Порівняння логічних виразів.
26. Способи та правила передачі аргументів у функцію.
27. Використання префіксної та постфіксної форм операторів ++ та --.
28. Оператор розгалуження та тернарний оператор, правила використання.
29. Оператори циклу, відмінності між ними та правила використання.
30. Оператор switch. Правила його використання.

Контрольні завдання

1. Чи правильно названі константи та змінні:

```
const int MIN_VALUE = 3;  
const double a1 = -5.8794;  
int b1 = 7;  
short countOfAnimals = 5;  
char s = 'x';
```
2. Напишіть програму, що виконує додавання, віднімання, множення та ділення двох чисел в діапазоні від -100 до 100. Які типи даних необхідно використати для зменшення обсягу використаної пам'яті?
3. Які назви функцій є правильними:

```
void prnTRSdssdc();  
int countOfElements();  
double smDvPnt();  
char getLetterFromArray();  
void pntRRR();
```
4. Напишіть програму для обчислення значення логічних виразів:

```
bool a = true;  
bool b = false;  
bool result1 = a and b or not a or not b and a;  
bool result2 = b and not a and not b and not a or b;  
bool result3 = not(a or b or not a or not b);
```

5. Перепишіть даний вираз з використанням оператора if-else


```
a > b ? cout << "a < b" : a > b ? cout << "a > b" : cout << "a == b".
```
6. Використовуючи тернарний оператор, напишіть програму для визначення стану води відповідно до заданої температури.
7. Виправте згідно з вимогами до гарного стилю програмування:


```
if (a == 0)
{
    cout << "a == 0";
}

double result = c / a + b;

if (x == 0 || y < 0 || z > 0)
{
    result = x + y + z;
}

long long time_of_lunch = 15;
unsigned short = -15;
```
8. Напишіть програму, що заповнює масив цілими числами та поелементно виводить елементи масиву у консоль. Використайте різні типи циклів.
9. Напишіть назви у CamelCase, PascalCase, ShakeCase:


```
myfunction1,      variable-for-testing,      messageforuser,      count-of-pages,
limitedvalueforcountofcycleiterations.
```
10. Розшифруйте значення імен:


```
MyClass.cpp, arrayOfAnimals, is_availalbe, SomeFunctions.h, total_count.
```
11. Проаналізуйте код лабораторних робіт з дисципліни «Основи програмування» з точки зору визначення імен файлів, констант, змінних, функцій, параметрів, типів. Відмітьте усі невідповідності вашого коду вимогам C/C++ Coding Standard. Перепишіть код з дотриманням вимог C/C++ Coding Standard. Зафіксуйте свою роботу по рефакторингу коду у звіті до лабораторної роботи.

3 Стиль документування та форматування коду

3.1 Повідомлення про авторські права

З юридичної точки зору програмний код вважається літературним твором і не має значення, що він написаний мовою програмування. Оскільки юридично програма — це літературний твір, на неї поширюються ті самі положення закону про авторське право і суміжні права, як і на літературні твори. Люди, що створили програмний код, вважаються його авторами або співавторами. Якщо колектив авторів великий, слід вказувати компанію, яка є володарем коду. Нижче поданий приклад оформлення коментаря про автора коду [5].

```
////////////////////////////////////
// ADOBE SYSTEMS INCORPORATED
// Copyright 2023 Adobe Systems Incorporated
// All Rights Reserved.
//
// NOTICE: Adobe permits you to use, modify, and distribute this file
// in accordance with the terms of the license agreement accompanying it.
////////////////////////////////////
```

3.2 Коментування

Усі програми мають бути прокоментовані таким чином, щоб легко описувати мету коду та будь-які алгоритми, які використовуються для досягнення мети. Коментування — це «мистецтво» опису того, що програма збирається робити, у висловлюваннях природною мовою «високого рівня». **Перш ніж писати код програми, спершу слід написати коментарі.**

Гарний код не вимагає пояснень у вигляді коментарів через застосування мнемонічних ідентифікаторів, дотримання принципів простоти, коректності та ясності, читабельності та зрозумілості коду. Зрозумілий код — це естетично гарний, структурований і не перевантажений код, який має єдине форматування, оптимальну структуру та логіку, такий, що легко тестується,

легко розширюється, має чіткі та компактні коментарі щодо логіки чи вимог конкретної функціональності, відповідає принципам KISS, DRY [3].

3.2.1 Типи коментарів, їх доцільність та основні правила

Можна виділити такі коментарі в коді: документаційні та пояснювальні.

Документаційні коментарі описують те, що робить функція, які аргументи приймає, що повертає, які винятки може викидати. Нижче поданий приклад оформлення коментаря документаційного коментаря:

```
/// <summary>
/// обчислює ряд Фібоначчі, використовуючи рекурсивний підхід
/// </summary>
/// <param name="n">кількість елементів ряду, що обчислюються</param>
/// <returns>значення елементів ряду Фібоначчі</returns>
int fibonacci(int n)
{
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Visual Studio оформлює документаційні коментарі у вигляді коментарів до документації XML. Вони починаються з потрібної косої лінії `///` і використовуються для опису фрагмента коду за допомогою XML-тегів. З таких коментарів згодом створюють окремі файли документації XML.

Пояснювальні коментарі використовуються всередині функціональних блоків і пояснюють, що відбувається в конкретних алгоритмах програми. Приклад пояснювального коментаря дивіться нижче:

```
int fibonacci(int n)
{
    //0 і 1 є вихідними значеннями
    if (n <= 1)
        return n;
    //виконати рекурсію для обчислення ряду Фібоначчі для заданого n
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Доцільність коментарів

Нижче поданий список типів коментарів, які варто писати в коді [4].

Copyright-коментарі в шапці файлів вихідного коду. Код завжди має автора і комусь належить, тому потрібно це вказувати.

Коментування інтерфейсу, що включає інтерфейс класу з описом публічних методів, опис глобальних констант, якщо в проєкті такі є, опис функцій. Будь-яка функціональність, яку використовують ваші колеги з команди розробки, повинна документуватись так, щоб було зрозуміло, що роблять конкретні функція або метод.

Опис та пояснення алгоритмів. У випадку використання складних математичних алгоритмів фахівці рекомендують використовувати коментарі для пояснення кроків виконання функції, для пояснення значень за замовчуванням для змінних та інших деталей коду, що допоможе його зрозуміти.

Коментарі TODO/FIXME для процесу рефакторингу. Рефакторинг — процес редагування програмного коду, внутрішньої структури програмного забезпечення з метою зробити його більш читабельним, чистим та зручним для розробників. Для полегшення розуміння коду та внесення подальших правок без зміни зовнішньої поведінки самої системи потрібно дати опис завдань, також описати, що потрібно зробити на конкретних ділянках коду в процесі рефакторингу.

Не варто писати такі коментарі:

- **коментарі без змісту.** Наприклад, не має змісту коментар перед ініціалізацією змінної «Variable initialization»;

- дублювання коміт-повідомлень у заголовках програмного коду. У більшості компаній вже використовується Git, що може відображати коментарі, зміни та все інше, що стосується вашого коду. Тому не пишіть такі коментарі;
- коментарі, що пояснюють імена функцій та змінних. Наприклад, змінна називається «mD». Тут краще дати таку назву змінної, що пояснюватиме її призначення, а не додавати коментар. Гарний код – це самодокументуючий код.

Основні правила написання коментарів [4]

1. Коментарі не повинні дублювати код.
2. Гарні коментарі не виправдовують незрозумілий код.
3. Якщо ви не можете написати чіткий коментар, проблема може бути в коді.
4. Коментарі мають розсіювати плутанину, а не викликати її.
5. Надайте посилання на оригінальне джерело скопійованого коду.
6. Додавайте посилання на зовнішні посилання, де вони будуть найбільш корисними.
7. Додавайте коментарі під час виправлення помилок.
8. Використовуйте коментарі, щоб позначити незавершені реалізації.

3.2.2 Коментування файлів

Використання системи контролю версій є рекомендованою практикою, але може стати втомливим завжди звертатися до системи контролю версій щодо змін, які відбуваються в базі коду. Стандарт гарного стилю програмування рекомендує, щоб заголовні та вихідні файли містили заголовок коментаря з описом функцій та призначення модуля. Коментар до файлу має включати такі дані [6]:

- Ім'я файлу
- Автор
- Дата модифікації файлу
- Номер версії модуля
- Запланована ціль
- Інформація про авторські права
- Різні нотатки
- Інформація про перегляд

Приклад коментаря до файлу:

```

////////////////////////////////////
// Ім'я файлу: individWork.cpp
// Автор: Іванов Іван, гр. ІПЗ-15
// Дата модифікації файлу: 2023.01.25
// Номер версії модуля: version 3
// Запланована ціль: Smart City: показати шляхи покращення та спрощення
//                      управління містом, благоустрій міського середовища,
//                      забезпечення безпеки та підвищення якості життя містян.
// Інформація про авторські права: програма розробляється в рамках НДР №23765
//                      усі права захищені, копіювання та поширення без дозволу
//                      автора заборонено.
// Bugs: визначені логічні помилки – не врахована функціональна вимога формування
//                      карти криміногенної ситуації в місті
//TODO: додані функції, що аналізують криміногенну ситуацію в місті та її візуалізують.
////////////////////////////////////

```

3.2.3 Коментування сигнатури функції

Сигнатура функції у програмуванні — це сукупність імені функції та її формальних параметрів. Кожна функція в коді повинна мати коментар. Такі коментарі відносяться до документаційних коментарів і описують семантику функції (що вона має виконати), параметри, виняткові ситуації, значення, що повертаються тощо. У заголовних (header) файлах коментарі до функцій призначені для користувачів інтерфейсу. Приклад застосування документаційних коментарів до сигнатури функції

```

/*
 * Функція бінарного пошуку у відсортованому масиві
 * Параметри:
 *   arr[] - одновимірний масив для пошуку
 *   size - розмір масиву
 *   left - ліва межа масиву
 *   right - права межа масиву
 *   keyValue - шукане значення
 * Значення, що повертаються:
 *   flag - ознака наявності чи відсутності шуканого значення
 */
void BinSearch(int arr[], int size, int left, int right, int keyValue, bool& flag)
{
    if (left > right)
        flag = false;
    else
    {
        middle = (left + right) / 2;
        if (keyValue == arr[middle])
            flag = true;
        else
            if (keyValue < arr[middle])
                BinSearch(arr, size, left, middle - 1, keyValue, flag);
            else BinSearch(arr, size, middle + 1, right, keyValue, flag);
    }
}

```

Доцільно скористатися рекомендаціями C Code Style Guidelines [5] щодо написання коментарів до функції: **спочатку слід написати коментар до функції, а потім вже код функції.**

3.2.4 Коментування локальних змінних функції

Коментарі мають бути корисними високорівневими описами того, що робить програма. Вони не повинні повторювати те, що є очевидним. Використовуючи відповідні імена змінних, більшу частину програми можна читати майже так само легко, як текст англійською мовою.

Код має бути самодокументований. У такому коді використовуються добре підібрані назви змінних і назви функцій, щоб код легко читався. Наприклад, визначення імен змінної а не має сенсу, але визначення імен змінної assigasy дає набагато кращий опис того, що має містити змінна. Використовуючи правильні імена змінних і функцій, потрібно звести до мінімуму кількість необхідних документаційних та пояснювальних коментарів. Проте стандарт гарного стилю програмування радить усі визначення імен в коді (змінні та функції) коментувати, визначаючи тим самим їх семантику [6].

Приклад коментування локальних змінних функції

```

/* Функція формування з одного вихідного масиву
 * двох масивів простих і складених чисел
 * Параметри:
 *   масив натуральних чисел
 * Значення, що повертаються:
 *   функція не повертає значення
 */
void simplenumber(int mas[])
{
    int divisor;           //дільник чисел масиву
    int arrsimple[10],      //масив простих чисел
        arrnotsimple[10];  //масив складених чисел
    int countsimple = 0,   //кількість простих чисел
        countnotsimple = 0; //кількість складених чисел
    for (int i = 0; i < n; i++)
    {
        /* перебір елементів вихідного масиву
         * визначення простих і складених чисел
         * підрахунок їх кількості

```

```

    формування масивів простих і складених чисел */
}
}

```

3.2.5 Коментування рішення (тіла функції)

Покращити читабельність коду можна за рахунок гарно написаним коментарям, які допомагають розробникам зрозуміти призначення, функціональність і складні деталі коду без необхідності розшифровувати його виключно на основі самого коду [4].

Кожен коментар до коду має служити певній меті та додавати цінність.

1. Використовуйте коментарі, щоб прояснити складні алгоритми обчислення або складні процеси прийняття рішень, наприклад:

```

// Множення матриці за допомогою алгоритму Штрассена
result = strassen_multiply(matrix1, matrix2);

```

2. Якщо ви робите якісь припущення під час написання коду, чітко вкажіть їх у коментарях, наприклад:

```

//Вважається, що вхідний масив відсортовано за зростанням
void binarySearch(array, target)
{
    //
}

```

3. Опишіть ширший контекст розділу коду, наприклад, його зв'язок з іншими модулями або його роль у загальній архітектурі програми. Це допомагає користувачам коду швидко зрозуміти мету та процес кодової бази:

```

// Функція обробляє мережевий запит і аналізує відповідь системи
void getUserData()
{
    // ...
},

```

4. Попереджайте про потенційні проблеми або потенційні підводні камені в коді, наприклад:

```

// Попередження: цей метод може спричинити витік пам'яті,
//якщо його не утилізувати належним чином
void InitializeResource()
{
    //
}

```

5. Обмежте свої коментарі розумною довжиною рядка, як правило, до 80 символів, щоб забезпечити читабельність на різних пристроях і редакторах коду.

6. Розміщуйте коментарі над кодом, на який вони посилаються, а не всередині. Це полегшує пошук коментарів під час прокручування файлів коду, наприклад,

```

// Перевірити введені користувачем дані
if (isValidInput(input))
{
    //TODO
}

```

7. Ясність, чіткість і стислість коментарів мають вирішальне значення для ефективного спілкування.

8. Уникайте надмірності: коментарі мають доповнювати код, а не дублювати його. Уникайте перефразування коду в коментарях; натомість надайте додаткову інформацію, яку неможливо отримати лише з коду. Наприклад, немає сенсу коментар:

```

//Збільшення лічильника
count++;

```

9. Регулярні оновлення коментарів при зміні коду. Застарілі коментарі можуть ввести розробників в оману та викликати плутанину. «Застарілі коментарі гірші, ніж відсутність коментарів» (Douglas Crockford).

Приклад коментарів алгоритму, що реалізований у функції, поданий нижче.

```
//Реалізації рекурсивного алгоритму пошуку вглиб у графі
//Параметри: V - показник на початкову вершину в графі
//Значення, що повертаються: функція не повертає значення
void Search(PtrVer V)
{
    if (V->number == finish)        //якщо досягнуто кінцевої вершини
    {
        cout << " Path: >> ";      //вивести пошуковий маршрут
        outputPath();
    }
    else                            //якщо кінцевої вершини не досягнуто
    for (int i = 0; i < V->k; i++)    //переглянути усі суміжні вершини
    {
        if (!(V->vertex[i]->mark)) //якщо поточна вершина не помічена
        {
            V->vertex[i]->mark = true; //помітити вершину та
            j++;                        //збільшити лічильник помічених вершин
            w[j] = V->vertex[i]->number; //запам'ятати номер поміченої вершини
            Search(V->vertex[i]);       //перейти до наступних вершин
            j--;
            V->vertex[i]->mark = false; //зняти мітку з пройденої вершини
        }
    }
}
```

3.2.6 Документування відповідей на запитання: що, як, чому

На рівні бібліотек/програм/функцій коментарі відповідають на питання «**Що?**»: «Що роблять ці бібліотеки/програми/функції?». Наприклад:

```
//Програма обчислює оцінку студента за навчальний семестр на основі його оцінок за модулі
//Функція використовує метод Ньютона для обчислення кореня функції
// Блок коду генерує випадкове число
```

Усередині бібліотек/програм/функцій коментарі відповідають на питання «**Як?**»: «Як код виконує завдання?». Наприклад,

```
/* Для обчислення підсумкової оцінки студента потрібно додати всі його оцінки за заняття і домашні завдання, потім поділити число, яке отримали, на загальну кількість оцінок. Таким чином, отримуємо середній бал студента.*/
```

На рівні операторів коду коментарі відповідають на питання «**Чому?**»: «Чому код виконує завдання саме так, а не інакше?». Поганий коментар на рівні операторів коду пояснює, що виконує код. Якщо код настільки складний, що потрібен коментар, який пояснює, що він виконує, в такому випадку потрібно не писати коментар, а переписувати код. Наприклад,

```
//Потрібно розділити всі елементи на 2, тому що вони куплені парами
cost = items / 2 * storePrice;
```

3.2.7 Коментування під час кодування

Є декілька причин, чому потрібно в певних ситуаціях закоментувати код [6]:

1. **Запуск програми, що не є налагодженою.** Потрібно запустити програму, яка поки що не є робочою. Компілятор не дозволить виконати програму, якщо в ній присутні помилки. Коментування неробочого коду відокремить його від робочого і дозволить запустити програму. Коли код буде робочий, його легко розкоментувати і продовжити роботу.

2. **Блокування логічних помилок.** Розроблений код, який компілюється, але має логічні помилки і працює не так як потрібно. Закоментуйте код з логічними помилками, виправте в ньому помилки, потім розкоментуйте його.

3. **Пошук кореня помилки.** Якщо програма формує результати, які не відповідають вимогам замовника, потрібно по черзі блокувати частини коду, щоб зрозуміти які з них робочі, а які є проблемними. Блокування частин коду можна за допомогою коментування. Якщо закоментувати такі рядки коду, щоб програма почала коректно працювати, ймовірно, що закоментовані рядки коду і є джерелом помилок.

4. **Тестування нового коду.** Замість видалення старого коду, ви можете його закоментувати і залишити для довідки, поки не будете впевнені, що ваш новий код працює так, як потрібно. Як тільки ви будете впевнені в новому коді, то зможете без проблем видалити старі фрагменти коду. Якщо ж новий код у вас працюватиме не так, як потрібно, то ви зможете його видалити і відновити старий код.

3.3 Форматування

На відміну від інших мов програмування, зокрема Python, мова C++ не має якихось обмежень у форматуванні коду з боку програмістів. Основне правило полягає лише в тому, щоб використовувати тільки ті способи, які максимально покращують читабельність і логіку коду.

IDE Visual Studio дозволяє візуалізацію структурування і автоматичне форматування коду в текстовому редакторі C/C++. Налаштування параметрів форматування здійснюється в діалоговому вікні (рис. 4).

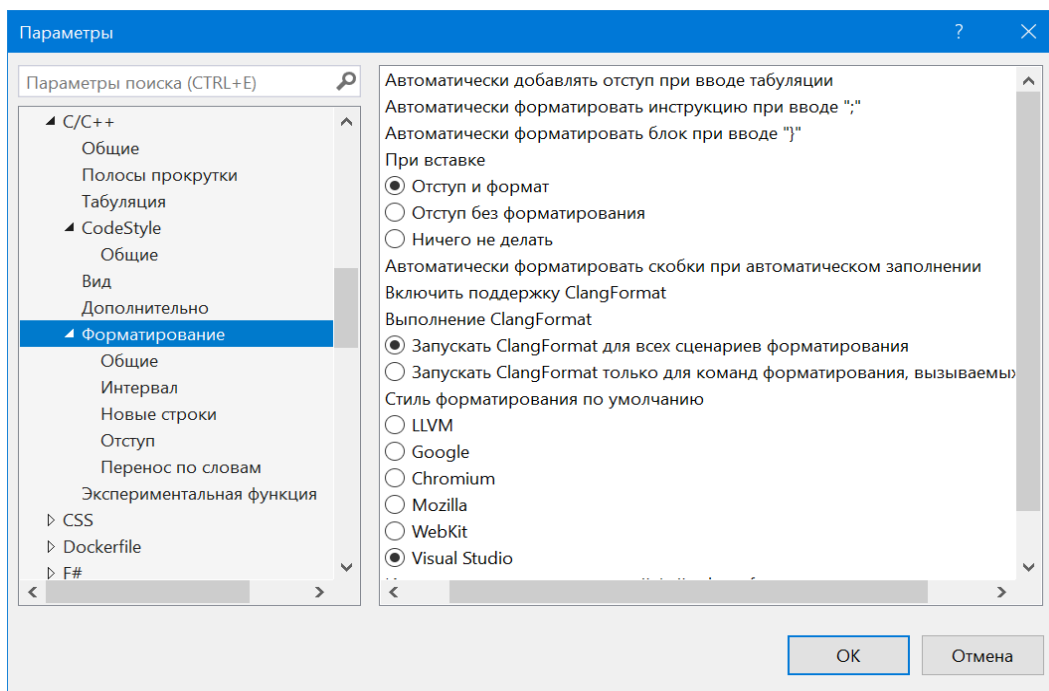


Рисунок 4. – Діалогове вікно «Параметри» для налаштування стилю форматування та візуального структурування коду

Основні параметри автоматичного форматування здійснюються при введенні операторів коду, блоку {}, при натисканні клавіші Enter, при вставленні тексту. В редакторі «VS Code» можна легко форматувати код або за допомогою комбінації клавіш Shift+Alt+F (діє на весь код у поточному файлі), або за допомогою комбінації клавіш Ctrl+K Ctrl+F (діє лише на код, що входить до поточного виділення). Обидва ці варіанти також доступні через контекстне меню, яке викликається клацанням правої кнопки миші (пункт «Форматувати документ», якщо клацнути в будь-якому місці коду, і пункт «Форматувати виділений фрагмент», якщо клацнути на виділеному фрагменті).

3.3.1 Політика фігурних дужок

Існує дві політики розміщення фігурних дужок. Перша політика, яка підтримується редактором «VS Code», розміщує фігурні дужки під ключовими словами в окремих рядках коду [7, 8]:

```
if (condition)
{

}
else
{
    ....
}
while (condition)
{
    ....
}
```

Редактор «VS Code» розміщує фігурні дужки строго по вертикалі, відмічаючи їх пари пунктирними лініями (рис. 5)

```
void copyarray()
{
    for (int i = 0; i < n; i++)
    {
        arrcopy[i] = arr[i];
    }
    outputArray(arrcopy, n);
}
```

Рисунок 5. – Розміщення фігурних дужок в редакторі «VS Code»

Політика Unix розміщує початкову фігурну дужку в одному рядку з ключовим словом, а кінцеву фігурну дужку – в окремому рядку з ключовим словом:

```
if (condition) {
    ...
}
else {
    ....
}
while (condition) {
    ....
}
```

Цей стиль не підтримується і автоматично не форматується редактором «VS Code». Отже, якщо ви використовуєте редактор Visual Studio, який підтримує зіставлення фігурних дужок, перший стиль є **набагато кращим**. Обґрунтування переваги першого стилю полягає в тому, що полегшується читання коду, оскільки не потрібно шукати початкову дужку, щоб знати, де починається відповідний блок коду. Особливо це важливо у випадку великих блоків, які містять декілька вкладених блоків. Виділивши курсором першу дужку “{”, ви легко знайдете відповідну їй дужку “}”, яку редактор коду виділить сам. Щоб переходити від блоку до блоку, вам просто потрібно використовувати курсор вниз і клавішу відповідності фігурної дужки. Немає необхідності рухатися до кінця рядка, щоб знайти дужку, а потім рухатися вперед і назад.

Гарним стилем програмування рекомендовано використовувати фігурні дужки в операторах if, while, do, for, навіть, якщо в дужках є лише один оператор. Це забезпечує безпеку коду під час додавання нових рядків, зберігаючи компактну форму для читання.

Також гарний стиль програмування рекомендує додавання коментарів до закриваючих дужок, що спростує розуміння коду, оскільки користувачу не потрібно шукати початкову дужку.

```
while (condition)
{
    if (valid)
    {

    } // if valid
}
```

```

else
{

} // not valid
// end forever
}

```

3.3.2 Політика відступів, табуляцій, пробілів

Зазвичай стандарт оформлення коду описує [2, 7-10]:

- засоби вибору значень та реєстр символів, що використовується, для будь-яких ідентифікаторів;
- запис типу змінної в її ідентифікаторі (угорська нотація або «верблюжа» нотація);
- реєстр символів (нижній, верхній, «верблюжий», «верблюжий» з малої букви);
- стиль відступів при оформленні логічних блоків (використання табуляції або іншої ширина відступу);
- спосіб розстановки дужок, що обмежують логічні блоки;
- використання пробілів при оформленні логічних та арифметичних виразів.
- стиль коментарів та використання коментарів, що документують методи, змінні чи класи.

Нагадаємо, що **угорська нотація імен** – це метод найменування ідентифікаторів у програмуванні, при якому до ідентифікатора змінної або функції додається префікс, що вказує на його тип. Наприклад, ідентифікатор `sClientName` рядкового типу має префікс `s`, ідентифікатор `iSize` має тип `int`, ідентифікатор `rBox` є покажчиком тощо.

Верблюжа нотація імен – це стиль написання складових слів, у якому кілька слів пишуться разом без пробілів, кожне слово усередині ідентифікатора пишеться з великої літери. Стиль отримав назву `CamelCase`, оскільки великі літери всередині слова нагадують горби верблюда (англ. *Camel*). Окремим випадком стилю є `lowerCamelCase`, коли з великої літери пишуться всі слова, крім першого. Наприклад, `RegularExpression`, `ConvertSpacesToTabs`, `visitNasaToday` (стиль `lowerCamelCase`) тощо.

Розглянемо політику відступів, що декларується стандартом гарного стилю програмування C/C++. Є два види відступів:

- горизонтальні відступи;
- вертикальні відступи.

Горизонтальний відступ робиться за допомогою двох або чотирьох пробілів, або за допомогою табуляції (клавіша `Tab`). Який відступ вибирати, вирішує програміст. Відступ з пробілами більш поширений на сьогодні. Однією з переваг пробілів є те, що пробіли дозволяють більш гнучку конфігурацію відступів, ніж табуляція.

Наприклад, ми можемо вирівняти параметри відносно відкритої дужки:

```

show(parameters,
    aligned, // 5 пробілів зліва
    one,
    after,
    another)
{
    // ...
}

```

Вертикальні відступи подаються пустими рядками для поділу коду на логічні блоки. Навіть окрема функція може бути поділена на логічні блоки. У наведеному нижче прикладі ініціалізація змінних, основний цикл та повернення результату виділені вертикальними відступами:

```

function exponentiation(x, n)
{
    int result = 1;
    //      <--
    for (int i = 0; i < n; i++)
    {

```

```

    result *= x;
}
//      <--
return result;
}

```

Рекомендації щодо відступів

1. Вставляйте додатковий рядок в тому випадку, коли це робить код зрозумілішим. Не повинно бути понад дев'яти рядків коду підряд без вертикального розподілу.
2. Зробіть відступ, використовуючи 3, 4 або 8 пробілів для кожного рівня.
3. Не використовуйте табуляції, використовуйте пробіли. Більшість редакторів можуть замінювати пробіли замість вкладок.
4. Табуляції мають бути зафіксовані на 8 проміжках. Не встановлюйте для табуляції інший інтервал, замість цього використовуються пробіли.
5. Робіть відступів стільки, скільки потрібно, але не більше. Немає довільних правил щодо максимального рівня відступів. Якщо рівень відступу перевищує 4 або 5 рівнів, слід подумати про те, щоб вилучити код.
6. Звичайні оператори повинні бути оточені пробілом :

```
a = (b + c) * d; // NOT: a=(b+c)*d;
```

7. Після зарезервованих слів C++ має стояти пробіл:

```
while (true) // NOT: while(true)
```

8. Після коми має бути пробіл:

```
doSomething(a, b, c, d); // NOT: doSomething(a,b,c,d);
```

9. Двокрапки повинні бути оточені пробілом:

```
std :: cout << // NOT: std::cout<<
```

10. Після крапки з комою в операторах for має стояти пробіл:

```
for (i = 0; i < 10; i++) // NOT: for(i=0;i<10;i++)
```

11. Логічні одиниці в блоці повинні бути розділені одним порожнім рядком.

Обґрунтування рекомендацій

Табуляція не використовується, оскільки відступ 8 пробілів сильно обмежує кількість рівнів відступу, які програма може мати. Реальний код часто може мати глибину трьох або більше рівнів. Зміна табуляції на менше, ніж 8 пробілів, недоцільна, оскільки це налаштування зазвичай є локальним і загальновизнаним для всіх редакторів. Ось чому стандарт гарного стилю програмування рекомендує використовувати пробіли.

Ніхто ніколи не може погодитися щодо правильної кількості пробілів, просто будьте послідовними. Загалом визначено, що 3 або 4 пробіли на рівень відступу є придатним.

```

void func()
{
    if (something bad)
    {
        if (another thing bad)
        {
            while (more input)
            {
            }
        }
    }
}

```

3.3.3 Політика круглих дужок

C++Coding Standard рекомендує таке застосування круглих дужок () з ключовими словами та функціями [5]:

- пишіть круглі дужки біля імен функцій, оскільки дужки () є оператором виклику функцій;

- не пишіть круглі дужки () зразу біля ключових слів. Поставте між ними пробіл. Ключові слова не є функціями, проте наявність круглих дужок () біля них візуалізує їх як назви функцій;
- не використовуйте круглі дужки () в операторах return, якщо це не потрібно.

Приклад

```
if (condition) // круглі дужки з ключовим словом if
{
}

while (condition) // круглі дужки з ключовим словом while
{
}

strcpy(s, s1); // круглі дужки для виклику функції
return result; // відсутність круглих дужок в операторі return
```

3.3.4 Довжина рядка коду

Рядки коду не повинні бути занадто довгими. Оптимальна довжина рядку коду 80 символів. Якщо рядок коду буде довший, його слід розбити на декілька окремих підрядків, наприклад [2]:

```
std :: cout << "Створити квадратну матрицю, розмірність якої n>=3 ввести з клавіатури."
<< " Передбачити вибір способу створення матриці: введення з клавіатури"
<< " або генерація псевдовипадкових чисел." << std :: endl;
```

Максимальна довжина рядка коду обґрунтована шириною рядка, який може друкувати принтер на аркуші A4.

Рядок коду може перевищувати 80 символів у випадках рядка коментаря, який неможливо розділити без шкоди для читання, легкості вирізання та вставлення чи автоматичного зв'язування, наприклад, якщо рядок містить літеральну URL-адресу, довжина якої перевищує 80 символів.

3.3.5 Форматування операторів розгалуження

Оператори розгалуження if-else або switch складаються з таких компонентів [7, 8]:

- одне або кілька ключових слів оператора (наприклад, if, else, switch);
- одна або декілька умов в круглих дужках;
- один або кілька контрольованих операторів або блоків контрольованих операторів у фігурних дужках {}.

Форматування цих операторів має дотримуватися таких правил гарного стилю програмування:

- компоненти операторів розгалуження повинні бути розділені одинарними пробілами (а не розривами рядків);
- усередині специфікатора умови має бути один пробіл (або розрив рядка) між кожною крапкою з комою та наступним токеном (лексемою), за винятком випадків, коли токен (лексема) є закриваючою дужкою або іншою крапкою з комою;
- у специфікаторі умови не має бути пробілу після відкриваючої або перед закриваючою дужкою;
- будь-які контрольовані оператори мають бути всередині блоків, які позначаються фігурними дужками {};
- усередині контрольованих блоків мають бути один розрив рядка відразу після відкриваючої дужки { та один розрив рядка безпосередньо перед закриваючою дужкою }.

Приклад ілюстрації означених вище правил для оператора if-else:

```
if (condition) // Гарний стиль – немає пробілів всередині круглих дужок,
               // є пробіл після if
{ // Гарний стиль – фігурна дужка блоку з нового рядка коду
  DoOneThing(); // Гарний стиль – відступи від початку блоку {}
```

```

    DoAnotherThing();
}
else // Гарний стиль - фігурна дужка } зіставлена з фігурною дужкою {
    if (a = f() && a != 3) // Гарний стиль - ключове слово else зіставлено з ключовим словом if
    { // Гарний стиль - фігурна дужка блоку { з відступом від початку
        DoAThirdThing(a); // свого блоку з нового рядка коду
    }
    else // Гарний стиль - ключове слово else зіставлено з відповідним
    { // ключовим словом if
        DoNothing();
    }
}

```

Приклад поганого стилю форматування оператора розгалуження if-else:

```

if(condition) {} // Bad - відсутній пробіл після if
else if ( condition ) {} // Bad - пробіл між дужками та умовою
else if ( condition){} // Bad - відсутній пробіл перед {
else if(condition){} // Bad - бракує кількох пробілів

```

До оператора поліваріантного вибору switch-case висуваються такі вимоги щодо його форматування:

- за ключовим словом switch у тому самому рядку має бути записаний у круглих дужках вираз, результатом якого є значення перелічуваного типу;
- кожне ключове слово case має відступ відносно ключового слова switch.
- перед символом : рекомендується вставити додатковий пробіл;
- явний коментар Fallthrough слід включати щоразу, коли є оператор case без оператора break. Відсутність оператора break є поширеною помилкою, і потрібно чітко пояснити, що це зроблено навмисно;
- default case завжди має бути присутнім і викликатися у випадку, якщо жоден з кейсів не відповідає виразу switch.

Приклад форматування оператора switch:

```

switch (condition)
{
    case ABC : statements; // Fallthrough
    case DEF : statements; break;
    case XYZ : statements; break;
    default : statements; break;
}

```

3.3.6 Форматування операторів циклу

Форматування операторів циклу має дотримуватися таких правил гарного стилю програмування [9]:

- компоненти операторів циклу повинні бути розділені одинарними пробілами (а не розривами рядків);
- усередині специфікатора ітерацій має бути один пробіл між кожною крапкою з комою та наступною лексемою, за винятком випадків, коли лексема є закриваючою дужкою або іншою крапкою з комою;
- у специфікаторі ітерації не має бути пробілу після відкриваючої або перед закриваючою дужкою;
- будь-які контрольовані оператори мають бути всередині блоків, які позначаються фігурними дужками {};
- усередині контрольованих блоків мають бути один розрив рядка відразу після відкриваючої дужки { та один розрив рядка безпосередньо перед закриваючою дужкою }.

Приклад ілюстрації означених вище правил для операторів циклу:

```

// Цикл з передумовою
while (condition)
{
    RepeatThing();
}

```

```

}
// Цикл з постумовою
do
{
    RepeatThing();
} while (condition);
// Цикл з лічильником – постфіксний інкремент
for (int i = 0; i < 10; i++)
{
    RepeatThing();
}
// Цикл з лічильником – префіксний інкремент
for (int i = 0; i < 10; ++i)
{
    RepeatThing();
}

```

В прикладах циклу `for` використовуються оператори інкременту/декременту версії префікс або постфікс. Якщо використовується префісний інкремент/декремент, значення змінної `i` спочатку збільшується/зменшується, а потім вже використовується. Але, якщо використовуються оператори інкременту/декременту версії постфікс, компілятор створює тимчасову копію змінної `i`, збільшує або зменшує оригінальну змінну `i` (НЕ копію), а потім повертає копію. Тільки після повернення копія змінної `i` видаляється.

Отже, можна вивести правило: використовуйте префісний інкремент і префісний декремент замість постфіксного інкременту і постфіксного декременту. Версії префікс не тільки продуктивніші, але і помилок з ними (за статистикою) менше.

Для простих скалярних значень (наприклад, параметрів циклу) немає причин віддавати перевагу префісний або постфіксній формі. Для ітераторів та інших типів шаблонів класів використовуйте оператори префіксної форми інкрементів та декрементів.

3.3.7 Один оператор або змінна – один рядок коду

Для покращення читабельності коду C++ Coding Standard рекомендує записувати у кожному рядку коду тільки один оператор [8]. Такі оператори зручніше коментувати у разі потреби. Також в процесі трасування коду їх легше аналізувати.

Також доцільно визначати одну змінну в рядку коду. Переваги такого визначення такі:

- для змінної в рядку можна додати документаційні коментар;
- змінну в рядку потрібно ініціалізувати;
- оголошення мають бути чіткими, що зменшує ймовірність оголошення змінної, коли насправді потрібно оголосити показник.

Приклад оголошення та ініціалізації змінних:

```

int    sizeArray;    // кількість елементів масиву
int*   arr = {0};    // покажчик на масив
int**  matrix;       // покажчик на матрицю
int    row = 0;       // кількість рядків матриці
int    column = 0;    // кількість стовпців матриці
float  profit = 1000; //прибуток
float& ref =profit;   //посилання на прибуток

```

Лексеми `*`, `**`, `&` мають бути записані поруч із типом, а не з ідентифікатором змінної.

3.3.8 Вирівнювання блоків оголошень

Усі блоки оголошень мають бути вирівняні, що робить код ясным і зрозумілим, таким, що легко читається та легко модифікується. Серед оголошень доцільно форматовувати блоки оголошень змінних та прототипів функцій, наприклад [7]:

```

//===== прототипи функцій
void Identification();    //ідентифікація автора
void taskProblem();       // умова задачі
void inputArray();        //введення масиву з клавіатури
void generateArray();     //генерація масиву випадковими числами
void outputArray(int[], int); //виведення масиву

```

```
//=====глобальні змінні
int min;                //мінімальний елемент
int minIndex;           //індекс мінімального
int max;                //максимальний елемент
int maxIndex;           //індекс максимального
```

Резюме

Усі програми мають бути прокоментовані таким чином, щоб легко описувати мету коду та будь-які алгоритми, які використовуються для досягнення мети.

Зрозумілий код — це естетично гарний, структурований і не перевантажений код, який має єдине форматування, оптимальну структуру та логіку, такий, що легко тестується, легко розширюється, має чіткі та компактні коментарі щодо логіки чи вимог конкретної функціональності.

Документаційні коментарі описують роботу функцій, які аргументи функція приймає, що функція повертає, які винятки може викидати.

Пояснювальні коментарі використовуються всередині функціональних блоків і пояснюють, що відбувається в конкретних алгоритмах програми.

Основні правила написання коментарів: коментарі не повинні дублювати код; гарні коментарі не виправдовують незрозумілий код; якщо програміст не може написати чіткий коментар, проблема може бути в коді; якщо код скопійований, слід надати посилання на оригінальне джерело; коментарі потрібні під час виправлення помилок в коді; використовуйте коментарі, щоб позначити незавершені реалізації.

Потребують коментування такі елементи коду: файли, сигнатура функцій, локальні та глобальні змінні, складні алгоритми обчислення або складні процеси прийняття рішень.

Коментарі мають відповідати на запитання «Що?», «Як?», «Чому?»

Максимально покращити читабельність і логіку коду можна за допомогою форматування коду, зокрема дотримання політики фігурних дужок, прийнятих в редакторі коду VS Code, який розміщує фігурні дужки {} під ключовими словами в окремих рядках коду, дотримання політики відступів, табуляцій, пробілів з використанням горизонтальних та вертикальних відступів, застосування круглих дужок () з ключовими словами та функціями, форматування операторів, вирівнювання блоків оголошень.

Контрольні запитання

1. Який код можна назвати зрозумілим?
2. Обґрунтуйте потребу в авторській інформації до коду?
3. Що таке коментар до коду?
4. Навіщо потрібні коментарі в коді?
5. Які типи коментарів існують?
6. Які елементи коду потрібно обов'язково коментувати?
7. Що таке сигнатура функції, і як вона має бути задокументована в коді?
8. Які основні правила написання коментарів означені в C++ Coding Standard?
9. Обґрунтуйте тезис: «Коментарі мають відповідати на запитання «Що?», «Як?», «Чому?»»
10. Назвіть причини, чому потрібно в певних ситуаціях закоментувати код, не видаляючи його?
11. Назвіть основні елементи коду, до яких слід застосовувати форматування коду.
12. Що розуміється під політикою фігурних та круглих дужок в коді C та C++?
13. Як застосування відступів, пробілів та табуляції покращують читабельність коду?
14. Які правила форматування операторів розгалуження та циклів слід застосовувати?
15. Чи є різниця між застосуванням префіксних або постфіксних операторів інкремента/декремента в операторі for?
16. Чому лексеми *, **, & мають бути записані поруч із типом, а не з ідентифікатором змінної?

Контрольні завдання

1. Проаналізуйте та виправте поданий нижче код:
 - а) вставте коментарі (до імені файлу, сигнатур функцій, локальних та глобальних змінних, алгоритмів тощо);
 - б) зробіть код самодокументованим;
 - в) переписіть код відповідно до вимог гарного стилю програмування

```
#include<iostream>
#include<time.h>
using namespace std;
const int maxn = 10000;
int arr[maxn][maxn];
int getRandom(int,int);
void inputArray(int ,int);
void inputKeyboard(int ,int);
void inputRandom(int,int);
void outputArray(int ,int);
void solve(int,int);
int main()
{
    srand(time(NULL));
    int m,n;
    cout << "Please, enter the size of the 2D array: " << endl;
    cin>> m >> n;
    inputArray(m,n);
    solve(m,n);
    return 0;
}
int getRandom(int l,int r)
{
    return l + rand()%(r-l+1);
}
void inputArray(int m,int n )
{
    cout << "Type 'y' for keyboard array input or type 'n' for random generation: ";
    char c;
    cin >> c;
    while(c != 'y' && c != 'n')
    {
        cout << "Incorrect data entry! Try entering 'y' or 'n'." << endl;
        cin >> c;
    }
    if(c == 'y')inputKeyboard(m,n);
    else inputRandom(m,n);
}
void inputKeyboard(int m,int n)
{
    cout << "Please, enter 2D array: ";
    for(int i = 0; i < m;i++)
        for(int j =0; j < n;j++)cin >> arr[i][j];
}
void inputRandom(int m,int n)
{
    for(int i = 0; i < m;i++)
        for(int j = 0; j < n;j++)arr[i][j]= getRandom(-100,100);
}
void outputArray(int m,int n)
{
    for(int i = 0; i < m;i++)
    {
        for(int j = 0; j < n;j++)cout << arr[i][j] << " ";
        cout << endl;
    }
}
void solve(int m,int n)
{
}
```

```

int minAmongMax,mini,minj;
for(int j = 0; j < n;j++)
{
    int maxn = arr[0][j];
    int maxni=0,maxnj=j;
    for(int i = 0; i < m;i++)if( arr[i][j] > maxn)
    {
        maxn = arr[i][j];
        maxni = i;
    }
    if(j == 0 || maxn < minAmongMax)
    {
        minAmongMax = maxn;
        mini = maxni;
        minj = maxnj;
    }
    arr[m][j] = maxn;
}
cout << "Array before conversion: " << endl;
outputArray(m,n);
cout << "Array after conversion: " << endl;
outputArray(m+1,n);
cout << "The value of the minimum element among the maximum: " << minAmongMax << endl;
cout << "Indices of the minimum element among the maximum: " << mini << " " << minj << endl;
}

```

2. Що в поданому нижче коду не відповідає C++ Coding Standard?

```

int x, * y;
int* x, *y;
char * c;
const std::string & str;
float f = 1.f;
long double ld = -.5L;
double d = 1248e6;
while (condition);
while (condition) continue;

```

3. Чому поданий нижче код мовою C++ має ознаки поганого коду? Перепишіть код відповідно до вимог гарного стилю програмування.

```

// BAD EXAMPLE
int addAndMultiply(int x, int y)
{
    int sum = x + y;
    int product = x * y;
    cout << "Sum: " << sum << endl;
    cout << "Product: " << product << endl;
    return sum;
}

```

4. Поясніть недоліки поданого нижче коду і перепишіть його відповідно до вимог гарного стилю програмування.

```

// BAD EXAMPLE
if (temperature > 30) cout << "It's hot outside." << endl;
else cout << "It's not hot outside." << endl;

```

5. Проаналізуйте код лабораторних робіт з дисципліни «Основи програмування» з точки зору його форматування, документування та визначення імен файлів, констант, змінних, функцій, параметрів, типів. Відмітьте усі невідповідності вашого коду вимогам C/C++ Coding Standard. Перепишіть код з дотриманням вимог C/C++ Coding Standard. Зафіксуйте свою роботу по рефакторингу коду у звіті до лабораторної роботи.

Частина 2. Об'єктно-орієнтований стиль програмування мовою C#

4 Проєктування класів C#

Класи є основною одиницею коду в C++, C#, Java та інших мов об'єктно-орієнтованого програмування (ООП). У цьому розділі наведено основні правила, яких слід дотримуватися під час написання коду ООП мовою C#.

4.1 Види класів

Існує велика кількість різних видів класів, які служать для різних цілей і, значить, слідує різним правилам. Згідно з [2] можна виділити такі види класів.

Класи-значення, наприклад для C++, `std::vector`, `std::list` моделюють вбудовані типи. Ці класи мають такі властивості:

- містять відкриті деструктор, конструктор копії та присвоєння з семантикою значення;
- не мають віртуальних функцій, включаючи деструктор;
- призначені для використання як конкретні класи, але не як базові;
- зазвичай розміщуються у стеку або є безпосередніми членами іншого класу.

Базові класи є будівельними блоками ієрархії класів. Базовий клас має такі властивості:

- має деструктор, який є або відкритим та віртуальним, або захищеним та невіртуальним, а також конструктор копії та оператор присвоєння, які не є відкритими;
- визначає інтерфейс за допомогою віртуальних функцій;
- зазвичай об'єкти такого класу створюються динамічно в купі як частина похідного класу та використовуються за допомогою інтелектуальних покажчиків.

Класи властивостей є шаблонами, які несуть інформацію про типи. Клас властивостей має такі характеристики:

- містить лише оператори `typedef` (для C++) та статичні функції. Клас не має модифікованого стану або віртуальних функцій;
- зазвичай об'єкти цього класу не створюються (конструктори можуть бути заблоковані).

Класи стратегій (зазвичай шаблони) є фрагментами змінної поведінки. Класи стратегій мають такі властивості:

- можуть мати стан та віртуальні функції, але можуть і не мати їх;
- зазвичай об'єкти даного класу не створюються, і він виступає як базовий клас чи член іншого класу.

Класи винятків поєднують семантику значень та посилання. При генерації винятків вони передаються за значенням, але мають перехоплюватися за посиланнями. Класи винятків мають такі властивості:

- мають відкритий деструктор та конструктори, які не генерують винятків, особливо конструктор копії, генерація виключення в якому призводить до завершення роботи програми;
- мають віртуальні функції та часто реалізують клонування;
- рекомендується їх робити похідними від `std::exception` (наприклад, для C++).

Допоміжні класи зазвичай підтримують окремі ідіоми. Важливо, щоб їх коректне використання не було пов'язане з труднощами і застосовувати їх некоректно було б дуже важко.

Гарний стиль програмування класів вимагає від розробника чіткої уяви про те, який вид класу розробник створює.

4.2 Принципи об'єктно-орієнтованого програмування

В парадигмі об'єктно-орієнтованого програмування визначено чотири основні принципи [9]:

Абстракція. Цей принцип моделює відповідні атрибути та взаємодію сутностей як класів для визначення абстрактного подання системи. Абстракція означає виділення основних, найбільш значущих показників предмета і відкидання другорядних, незначних.

Інкапсуляція. Цей принцип забезпечує приховування внутрішнього стану та функціональності об'єкта та надання доступу до них лише через загальнодоступний набір методів (функцій).

Успадкування означає можливість створювати нові абстракції на основі існуючих абстракцій. Цей принцип дозволяє створювати нові класи на основі наявних (батьківських класів), з можливістю перевизначення або доповнення їх властивостей та методів.

Поліморфізм означає здатність реалізувати успадковані властивості або методи різними способами в кількох абстракціях.

Крім цих принципів Роберт Мартін, Барбара Лісков і Бертран Меєр визначили ще п'ять принципів об'єктно-орієнтованого проєктування, які відомі як SOLID принципи і є важливою складовою мислення розробника ООП.

Принцип єдиного обов'язку (Single Responsibility Principle) свідчить про те, що кожен клас повинен виконувати лише один обов'язок. Це означає, що всі методи класу мають бути сфокусовані на виконання одного спільного завдання. Якщо є методи, які не відповідають меті створення класу, їх треба винести за його межі.

Принцип відкритості/закритості (Open/Close Principle) означає, що класи мають бути відкриті для розширення, але закриті для змін. Суть цього принципу полягає в тому, що програма будується таким чином, що її подальші зміни повинні бути реалізовані за допомогою додавання нового коду, а не зміни вже існуючого.

Принцип підстановки Лісков (Liskov Substitution Principle) Клас, що успадковує базовий клас, має доповнювати, а не замінювати поведінку базового класу. Принцип означає, що у випадку заміни об'єкту базового класу об'єктом його похідного класу, програма має продовжувати працювати коректно.

Принцип розділення інтерфейсу (Interface Segregation Principle) означає, що клієнти не повинні залежати від методів, які вони не використовують, тобто занадто «товсті» інтерфейси необхідно розділяти на менші та специфічні, щоб їх клієнти знали лише про ті методи, що необхідні для них у роботі.

Принцип інверсії залежностей (Dependency Inversion Principle), який складається з двох тверджень:

- модулі вищого рівня не повинні залежати від модулів нижчого рівня. Обидва типи модулів повинні залежати від абстракцій;
- абстракції не повинні залежати від деталей реалізації. Деталі реалізації повинні залежати від абстракцій.

Принципи SOLID вирішують проблеми невдалого проєктування програм. Показники невдало спроектованої програми такі:

- у програму, або її частину, складно внести зміни, оскільки будь-яка зміна впливає на занадто велику кількість компонентів системи, тобто програма є **жорсткою**;
- при внесенні змін порушується робота компонентів системи у несподіваних місцях, тобто програма є **крихкою**;
- складно використати той самий код в іншій програмі, оскільки його неможливо витягти з даної програми, тобто програма є **непорушною**.

4.3 Структура класів

Члени типу (класу) включають усі методи, поля, константи, властивості та події. У C# немає глобальних змінних або методів, як у деяких інших мовах. Навіть точка входу програми, метод `Main()`, має бути оголошений у класі чи структурі (неявно, коли ви використовуєте оператори верхнього рівня).

Наведений нижче список (табл. 2) включає всі типи членів, які можуть бути оголошені в класі, структурі або записі [13].

Таблиця 2. – Перелік членів класу C#

Члени класу	Опис
Поля	Поля — це змінні, оголошені в області видимості класу. Поле може бути вбудованим числовим типом або екземпляром іншого класу. Наприклад, клас календаря може мати поле, яке містить поточну дату.
Константи	Константи — це поля, значення яких встановлюються під час компіляції та не можуть бути змінені.
Властивості	Властивості — це методи класу, доступ до яких здійснюється так, ніби вони є полями цього класу. Властивість може забезпечити захист поля класу, щоб уникнути його змін без відома об'єкта.
Методи	Методи визначають дії, які може виконувати клас. Методи можуть приймати параметри, які надають вхідні дані, і можуть повертати вихідні дані через параметри. Методи також можуть повертати значення безпосередньо, без використання параметра.
Події	Події надають сповіщення іншим об'єктам про випадки, такі як натискання кнопок або успішне завершення методу. Події визначаються та запускаються за допомогою делегатів.
Оператори	Перевантажені оператори вважаються членами типу. Коли ви перевантажуєте оператор, ви визначаєте його як публічний статичний метод у типі. Щоб отримати додаткові відомості, див. Перевантаження оператора.
Індексатори	Індексатори дозволяють індексувати об'єкт подібно до масивів.
Конструктори	Конструктори — це методи, які викликаються під час першого створення об'єкта. Вони часто використовуються для ініціалізації даних об'єкта.
Фіналізатори	Фіналізатори дуже рідко використовуються в C#. Це методи, які викликаються механізмом виконання, коли об'єкт збирається видалити з пам'яті. Зазвичай вони використовуються, щоб переконатися, що будь-які ресурси, які потрібно звільнити, обробляються належним чином.
Вкладені типи	Вкладені типи — це типи, оголошені в іншому типі. Вкладені типи часто використовуються для опису об'єктів, які використовуються лише типами, які їх містять.

Доступність членів класу

Деякі методи та властивості призначені для виклику або доступу з коду поза класом або структурою, відомого як код клієнта. Інші методи та властивості можуть використовуватися лише в класі або самій структурі. В мові C# ви вказуєте, наскільки ваші типи та їхні члени доступні для клієнтського коду, використовуючи такі модифікатори доступу [13]:

- public;
- protected;
- internal;
- protected internal;
- private;
- private protected.

Відповідно до гарного стилю програмування за C# Coding Standards оголошення класу має починатися зі списку полів, тобто атрибутів (змінних), в яких зберігаються значення властивостей об'єктів.

Спочатку перераховуються відкриті статичні константи. Далі слідують приватні статичні змінні, а за ними йдуть приватні змінні екземплярів. Відкритих змінних зазвичай немає, важко знайти вагомі причини для їх використання.

За списком атрибутів (змінних) зазвичай слідують відкриті методи (функції). Автори рекомендують розміщувати приватні допоміжні методи (функції), що викликаються відкритими методами (функціями), безпосередньо за відкритою функцією.

Приклад оголошення класу мовою C#:

```
class Student
{
    /// <summary>
    ///   поля (атрибути) класу
    /// </summary>
    private int id = 0;           // ідентифікатор користувача
    private string login = string.Empty; // логін користувача
    /// <summary>
    ///   конструктор класу
    /// </summary>
    public Student(int studId, string studLogin)
    {
        id = studId;
        login = studLogin;
    }
    /// <summary>
    ///   властивість для поля id класу
    /// </summary>
    public int StudentId
    {
        get { return id; }
        set
        {
            if (value > 0)
                id = value;
        }
    }
    /// <summary>
    ///   властивість для поля login класу
    /// </summary>
    public string StudentLogin
    {
        get { return login; }
        set
        {
            login = value;
        }
    }
    /// <summary>
    ///   метод класу - вивід даних про користувача на консоль
    /// </summary>
    public void GetStudentDetails()
    {
        // прямий доступ до полів id, name класу
        Console.WriteLine("Id: {0}, Login: {1}", id, login);
        // доступ до полів id, name через властивості класу UserId, UserName
        Console.WriteLine("Id: " + StudentId + ", Login: " + StudentLogin);
    }
    /// <summary>
    ///   завдання нового логіну студента
    /// </summary>
    /// <param name="login">новий логін студента</param>
    /// <returns> нове значення поля login </returns>
    public string SetLogin(string newlogin)
    {
        if (IsValidLogin(newlogin))
        {
            login = newlogin;
        }
        else
        {
            Console.WriteLine("не введений новий логін");
        }
        return login;
    }
    /// закритий метод перевірки логіна
    /// <param name="nick">логін, що перевіряється</param>
    /// <returns> повертається true, якщо заданий рядок,
    ///   повертається false, якщо рядок пустий</returns>
    private bool IsValidLogin(string nick)
    {
        return !string.IsNullOrEmpty(nick);
    }
} // end of class Student
```

4.4 Мінімальні або компактні класи

Код програмної системи вважається гарним і відповідає вимогам гарного стилю програмування, якщо програмна система складатися з **безлічі дрібних класів**, а не з невеликої кількості великих.

Згідно з C++ Coding Standards [2]:

- мінімальний або компактний клас втілює одну концепцію на рівні деталізації;
- монолітний або великий клас зазвичай включає кілька окремих концепцій, і використання лише однієї з них спричиняє зайві накладні витрати ;
- мінімальний клас легше зрозуміти та простіше використовувати у тому числі повторно;
- мінімальний клас простіше у використанні. Монолітний клас часто має використовуватися як велике неподільне ціле. Наприклад, монолітний клас `Matrix` реалізує і використовує метод обчислення власних значень матриці. Але більшості користувачів цього класу потрібні лише ази лінійної алгебри. Найкращим варіантом буде реалізація різних функціональних областей у вигляді функцій, які не є членами класу `Matrix`, але працюють з мінімальним типом `Matrix`. Тоді ці функціональні області можуть бути протестовані та використані окремо лише тими користувачами, хто їх потребує;
- монолітні класи знижують інкапсуляцію. Якщо клас має багато методів (функцій-членів), які не повинні бути членами класу, проте є такими, тоді закриті атрибути (члени-дані) класу стають майже настільки ж поганими з погляду дизайну, як і відкриті змінні;
- монолітні класи зазвичай є результатом спроб передбачити та надати повне вирішення певної проблеми; на практиці ж такі дії майже ніколи не призводять до успішного результату;
- монолітні класи складніше зробити коректними та безпечними у зв'язку з тим, що при їх розробці найчастіше порушується принцип «**Один об'єкт – одне завдання**».

4.5 Принцип єдиної відповідальності

Кожен клас інкапсулює одну відповідальність, має одну причину для зміни та взаємодіє з іншими класами для реалізації бажаної поведінки системи. **Принцип єдиної відповідальності – одна з найважливіших концепцій в ООП.**

Як і у випадку з функціями, компактність і принцип єдиної відповідальності має стати основними правилами проєктування класів. Наведемо приклад порушення принципу єдиної відповідальності мовою C# [14]:

```
/* Є книга, яка може надавати інформацію про свою назву, автора
 * та здатна перегортати сторінки. Також цей клас може відображати
 * на екрані текст поточної сторінки книги.
 */
class Book
{
    public string getTitle()
    {
        return "A Great Book";
    }
    public string getAuthor()
    {
        return "John Doe";
    }
    public void turnPage()
    {
        // pointer to next page
    }
    public void printCurrentPage()
    {
        Console.WriteLine("current page content");
    }
}
```

В цьому класі порушений принцип єдиної відповідальності, бо насправді тут їх дві:

- управління книгою (наприклад, бібліотекар);
- механізм подання даних (наприклад, спосіб, за допомогою якого можна виводити вміст книги користувачеві: на екран, у графічному вигляді або друкувати тільки текст.

Поєднання бізнес-логіки з представленням є вкрай небажаним, оскільки це суперечить принципу єдиної відповідальності, тому слід розділити функціональність класу Book і подати її у вигляді двох класів, наприклад, класу книги і класу друку певної сторінки книги:

```
class NewBook
{
    public string getTitle()
    {
        return "A Great Book";
    }
    public string getAuthor()
    {
        return "John Doe";
    }
    public void turnPage()
    {
        // pointer to next page
    }
    public string getCurrentPage()
    {
        return "current page content";
    }
}
class TextPrinter
{
    public void PrintPage(int page)
    {
        Console.WriteLine("book content with number page " + page);
    }
}
```

5 Програмування класів в C#

5.1 Правила визначення імен

1. Використання PascalCase нотації

Згідно із C# Coding Standard [12] для назв **просторів імен, класів, структур, методів класів, властивостей класів, константних полів, публічних полів** класів використовується Паскаль-нотація (PascalCase). PascalCase – це стиль написання імен, при якому складні слова в назві ідентифікатора пишуться разом, і кожне слово розпочинається з великої букви. Наприклад, MyVar, MyBestProgramm, WorkArray. Використовуйте іменник або словосполучення з іменником, щоб назвати клас.

```
public class SomeClass
{
    private const int DefaultSize = 100;
    public string Name;
    public string PropertyNaming { get; set; }
    public void SomeMethod()
    {
        //TODO
    }
}
```

2. Використання CamelCase нотації

Визначення імен **аргументів методів, приватних полів і локальних змінних і констант** здійснюється з використанням верблюжої нотації (CamelCase). CamelCase – це стиль написання імен, при якому складні слова в назві ідентифікатора пишуться разом, але кожне зі слів, крім першого, починається з великої літери. Наприклад: firstElement, myField, lastName.

```
public void ProcessData(int itemCount)
{
    string customerName = "Ivanov C.";
    // Logic using itemCount and customerName
}
```

3. Використання угорської нотації

C# Coding Standard не рекомендує використовувати угорську нотацію для визначення імен програмних сутностей. Угорська нотація передбачає використання в якості префікса ідентифікатора літеру, що визначає тип змінної. Це робити **недоцільно**, оскільки Visual Studio IDE легко визначає тип змінної за допомогою підказок.

```
// Correct
int counter;
string name;
// Avoid
int iCounter;
string strName;
```

4. C# Coding Standard не рекомендує використовувати підкреслення в ідентифікаторах.

```
// Correct
public string clientAppointment;
public TimeSpan timeLeft;
// Avoid
public string client_Appointment;
public TimeSpan time_Left;
```

5. Використовуйте попередньо визначені імена типів замість імен системних типів, як-от Int16, String, UInt64 тощо.

```
// Correct
string firstName;
int lastIndex;
bool isSaved;
// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
```

6. Використовуйте неявний тип var для оголошень локальних змінних. Виняток: примітивні типи (int, string, double тощо) використовують попередньо визначені імена.

```
var stream = File.Create(path);
var customers = new Dictionary();
// Exceptions
int index = 100;
string timeSheet;
bool isCompleted;
```

7. Для визначення імен **інтерфейсів** використовуйте префікс I. Назви інтерфейсів – це іменники або словосполучення з іменником, або прикметники.

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

8. Із значення імен **вихідних файлів** має відповідати визначенню імен класів, які описані в цих файлах. Виняток: імена файлів із частковими класами відображають їх джерело або ціль, наприклад, дизайнер, згенерований тощо.

```
// Located in Task.cs
public partial class Task
{
    //...
}
// Located in Task.generated.cs
public partial class Task
{
    //...
}
```

9. Для параметрів циклів використовуйте ідентифікатори *i*, *j*, *k*, *l*, *m*.

10. Не скорочуйте терміни та не використовуйте акроніми, оскільки це призводить до неузгодженості використання коду та інших артефактів програми та погіршує читабельність. Зокрема, не скорочуйте «EducationOrganization» як «EdOrg» або «LocalEducationAgency» як «Lea». Винятки: загальновідомі терміни, такі як ODS, API, Id, msg, max, min, JSON тощо.

Приклади правил визначення імен усіх програмних сутностей C# подані в табл. 3.

Таблиця 3. – Стандарт визначення імен програмних сутностей C#

Програмна сутність	Нотація	Приклад
Namespace	PascalCase	System.Drawing
Class, struct	PascalCase	BankAccount
Interface	PascalCase	IShapeCollection
Константе поле	PascalCase	MaximumItems
Властивість	PascalCase	PropertyName
Метод	PascalCase	CalculateSum
Enum	PascalCase	ColorName
Елементи Enum	PascalCase	Red
Подія	PascalCase	SecurityAlert
Private static readonly поле	PascalCase	StudentDepartment
Non-private поле	PascalCase	StudentName
Тип параметру	PascalCase	TView
Private поле	CamelCase	arrayItem
Параметр методу	CamelCase	bookPage
Локальні змінні	CamelCase	sizeOfArray

5.2 Форматування коду C#

У наступних пунктах описані способи реалізації вихідного коду C#, щоб створити читабельний, зрозумілий і послідовний код, який легко зрозуміти та підтримувати. C# Coding Standards формулює такі правила форматування коду:

1. Згідно із C# Coding Standards [12-15] фігурні дужки {} повинні бути вертикально вирівняні. Це забезпечує кращу читабельність коду користувачами та легкість формування документації до коду.

```
// Correct
class Program
{
    static void Main(string[] args)
    {
    }
}
```

2. Оголошуйте лише одну змінну на рядок

```
// Bad
int counter = 0, index = 0;
// Good
int counter = 0;
int index = 0;
```

3. Додавайте порожні рядок після закриваючих дужок, якщо наступний оператор не є частиною продовжуваної мовної конструкції (наприклад, if / else, try / catch / finally), після списку using, між методом, властивостями та внутрішнім типом оголошень, в кінці кожного файлу.

```
// Good style
public static string ToCamelCase(this string text)
{
    if (string.IsNullOrEmpty(text))
    {
        return text;
    }
    return char.ToLower(text[0]) + text.Substring(1);
}
```

```
// Bad style
public static string ToCamelCase(this string text)
{
    if (string.IsNullOrEmpty(text))
    {
        return text;
    }
    return char.ToLower(text[0]) + text.Substring(1);
}
```

4. Автоматичні властивості розміщуйте в одному рядку, наприклад,

```
// Good style
public int Age { get; set; }
```

```
// Bad style
public int Age
{
    get;
    set;
}
```

5. Використовуйте дужки (), щоб зробити речення у виразі очевидними, як показано в наступному коді:

```
if ((val1 > val2) && (val1 > val3))
{
    // TODO. Take appropriate action.
}
```

6. Під час побудови багаторядкових логічних виразів ставте логічний оператор на початку кожного рядка, наприклад:

```
// Good
public bool IsEmpty()
{
    return
        !(EducationOrganizationId.HasValue || StateEducationAgencyId.HasValue
          || EducationServiceCenterId.HasValue || LocalEducationAgencyId.HasValue
          || SchoolId.HasValue || EducationOrganizationNetworkId.HasValue)
        && string.IsNullOrEmpty(StaffUniqueId)
        && string.IsNullOrEmpty(StudentUniqueId)
}

// Bad
public bool IsEmpty()
{
    return
        !(EducationOrganizationId.HasValue || StateEducationAgencyId.HasValue
          || EducationServiceCenterId.HasValue || LocalEducationAgencyId.HasValue
          || SchoolId.HasValue || EducationOrganizationNetworkId.HasValue) &&
        string.IsNullOrEmpty(StaffUniqueId) &&
        string.IsNullOrEmpty(StudentUniqueId) &&
}
```

7. Не вставляйте порожній рядок після відкриваючої дужки {, перед закриваючою дужкою }, після блоку коментарів, або однорядкового коментаря, поруч з іншим порожнім рядком.

8. Не пишіть конструктори без тіл, але, якщо ви визначили такий конструктор, скоротіть його до одного рядка, наприклад:

```
public Employee() {}
```

9. Записуйте ключове слово base в окремому рядку з відступом відповідно до прикладу:

```
public Employee(string name, int age, string company)
    : base(name, age)
{
    this.company = company;
}
```

10. Ніколи не оголошуйте більше ніж один простір імен в одному файлі.

11. Уникайте розміщення кількох класів в одному файлі.

12. Вставляйте пробіл навколо двійкового та тернарного операторів, між відкриваючою дужкою і ключовими словами `if`, `for`, `foreach`, `catch`, `while`, `lock`, або `using`, до і в межах однорядкового блоку доступу, між аксесорами в одному рядковому блоці аксесорів, після коми, яка не знаходиться в кінці рядка, після крапки з комою в виразі `for`, після двокрапки в одному рядковому виразі `case`, навколо двокрапки в оголошенні типу, навколо стріли лямбда. Після коментаря однорядкового символу (`//`) і перед ним, якщо він використовується в кінці рядка.

5.3 Коментування коду C#

Практично всі мови програмування дозволяють залишати у коді коментарі. Вони ніяк не використовуються кодом і потрібні виключно для людей, які користуватимуться кодом. Коментарі в C# бувають трьох видів [16]:

- однорядкові коментарі починаються з лексеми `//`. Після цих двох символів може слідувати будь-який текст, весь рядок не буде аналізуватись і виконуватись;

```
private int id = 0;      // ідентифікатор користувача
```

- багаторядкові коментарі починаються з лексеми `/*` та закінчуються лексемою `*/`. Прийнято кожен рядок починати із символу `*`, хоча технічно це не обов'язково;

```
/* За даними, що введені з консолі, визначити значення виразу,  
   використовуючи математичні функції, і вивести результат на консоль.  
*/
```

- Документуючі коментарі (XML коментарі) кожного рядка використовують лексему `///`. Документуючі коментарі, будучи по суті підвидом однорядкових, додають підказки до коду для зовнішніх користувачів коду.

```
/// <summary>  
/// властивість для поля login класу  
/// </summary>
```

C# Coding Standards формулює такі правила коментування коду [13-16]:

1. Усі коментарі мають бути написані однією мовою, бути граматично правильними та містити відповідні розділові знаки.

2. Для позначення коментарів в C# використовуйте `//` або `///`, але ніколи `/* ... */`

3. Не створюйте блоків коментарів типу «квітова коробка». Наприклад:

```
// *****  
// Блок коментарів  
// *****
```

4. Використовуйте коментарі, щоб пояснити припущення, відомі проблеми та інформацію про алгоритми. Не використовуйте коментарі для пояснення очевидного коду. Добре написаний код самодокументується.

5. Використовуйте коментарі до поганого коду, щоб сказати «виправте цей код», інакше видаліть або переписіть код!

6. Додавайте коментарі, використовуючи позначки ключових слів списку завдань, щоб дозволити фільтрацію коментарів. Наприклад:

```
// TODO: розмістити тут код бази даних  
// СКАСОВАНО: виклик Invoke видалено через помилки  
// НАСК: Тимчасове виправлення, доки не буде здійснено рефакторинг
```

7. Завжди застосовуйте документуючі коментарі C# у вигляді блоків `///` до публічних, захищених і внутрішніх декларацій. Для документування API використовуйте лише блоки коментарів C# (`///`).

8. Завжди включайте коментарі `<summary>`. Додавайте розділи коментарів `<param>`, `<return>` і `<exception>`, де це можливо.

```
/// <summary>  
/// завдання нового логіну студента  
/// </summary>
```

```

/// <param name="login">заданий новий логін студента</param>
/// <returns> повертається значення логіну як поле класу</returns>
public string SetLogin(string newlogin)
{
    if (IsValidLogin(newlogin))
    {
        login = newlogin;
    }
    else
    {
        Console.WriteLine("не введений новий логін");
    }
    return login;
}

```

Резюме

Гарний стиль програмування класів вимагає від розробника чіткої уяви про те, який вид класу розробник створює: класи-значення, базові класи, класи властивостей, класи стратегій, класи винятків, допоміжні класи.

У парадигмі об'єктно-орієнтованого програмування визначено чотири основні принципи: абстракція, інкапсуляція, успадкування, поліморфізм.

SOLID принципи об'єктно-орієнтованого проєктування: принцип єдиного обов'язку, принцип відкритості/закритості, принцип підстановки Лісков, принцип розділення інтерфейсу, принцип інверсії залежностей.

Члени типу (класу) включають усі константи, поля, властивості, методи та події. У C# немає глобальних змінних або методів, що не є членами класів. Навіть точка входу програми, метод Main(), має бути оголошений у класі чи структурі (неявно, коли ви використовуєте оператори верхнього рівня).

У мові C# ви вказуєте, наскільки ваші типи та їхні члени доступні для клієнтського коду, використовуючи такі модифікатори доступу: public, protected, internal, protected internal, private, private protected.

Код програмної системи вважається гарним і відповідає вимогам гарного стилю програмування, якщо програмна система складатися з безлічі дрібних класів, а не з невеликої кількості великих.

Мінімальний або компактний клас втілює одну концепцію на рівні деталізації.

Кожен клас інкапсулює одну відповідальність, має одну причину для зміни та взаємодіє з іншими класами для реалізації бажаної поведінки системи. Принцип єдиної відповідальності – одна з найважливіших концепцій в ООП. Як і у випадку з функціями, компактність і принцип єдиної відповідальності має стати основними правилами проєктування класів.

Згідно із C# Coding Standard [13] для назв просторів імен, класів, структур, методів класів, властивостей класів, константних полів, публічних полів класів використовується Паскаль-нотація (PascalCase), відповідно до якої кожне слово у складі ідентифікатора, розпочинається з великої букви.

Визначення імен аргументів методів, приватних полів і локальних змінних і констант здійснюється з використанням верблюжої нотації (CamelCase), відповідно до якої кожне зі слів, крім першого, починається з великої літери.

C# Coding Standard не рекомендує використовувати підкреслення в ідентифікаторах

Згідно із C# Coding Standards фігурні дужки {} повинні бути вертикально вирівняні. Це забезпечує кращу читабельність коду користувачами та легкість формування документації до

Уникайте розміщення кількох класів в одному файлі.

Завжди застосовуйте документуючі коментарі C# у вигляді блоків /// до публічних, захищених і внутрішніх декларацій. Завжди включайте коментарі <summary>. Додавайте розділи коментарів <param>, <return> і <exception>, де це можливо.

Контрольні запитання

1. Принципи об'єктно-орієнтованого програмування та їх сутність
2. Принципи SOLID та їх сутність.
3. Структура класу C#
4. Рекомендації щодо послідовності опису членів класу.
5. Поняття полів, властивостей і методів класу C#
6. Який стиль використовується для визначення імен просторів імен, класів, структур, методів класів, властивостей класів, константних полів, публічних полів класів?
7. Який стиль використовується для визначення імен аргументів методів, приватних полів і локальних змінних і констант?
8. З якою метою фігурні дужки {} повинні бути вертикально вирівняні?
9. Чому не рекомендується розміщення кількох класів в одному файлі?
10. Що таке документуючі коментарі?
11. Для чого використовують документуючі коментарі?

Контрольні завдання

1. У поданому нижче коду розставте пробіли, відступи та вставте документуючі коментарі згідно із C# Coding Standard:

```
public class MyClass<A, B>:Parent<A, B>
{
    public float MyProperty{get; set;}
    public float AnotherProperty
    {
        get{return MyProperty;}
    }
    public void MyMethod()
    {
        int[] values={1, 2, 3, 4};
        int sum=0;
        for (int i = 0; i<values.Length; i++) {
            switch (i){
                case 3:return;
                default:
                    sum+=i > 2 ? 0:1;
                    break;
            }
        }
        i+=(int)MyProperty;
    }
}
```

2. Виправте визначення імен сутностей класу відповідно до конвенцій визначення імен стилями PascalCase та CamelCase в поданому нижче коді:

```
namespace Exampleproject
{
    public class playerCharacter
    {
        public const float defaultSpeed = 10f;
        public float currentspeed { get; set; }
        protected int hit_points;
        private void calculate_WeaponDamage()
        {
        }
    }
}
```

3. Поданий нижче код має 7 ознак поганого стилю програмування C#. За якими ознаками поданий нижче код є поганим? Перерахуйте усі 7 ознак поганого стилю програмування C#, які є в даному коді, та переписіть його відповідно до C# Coding Standard.

```
public class Class1
{
    public decimal Calculate(decimal amount, int type, int years)
    {
        decimal result = 0;
        decimal disc = (years>5) ? (decimal)5/100 : (decimal)years/100;
        if (type == 1)
        {
            result = amount;
        }
        else if (type == 2)
        {
            result = (amount - (0.1m * amount)) - disc * (amount - (0.1m * amount));
        }
        else if (type == 3)
        {
            result = (0.7m * amount) - disc * (0.7m * amount);
        }
        else if (type == 4){
            result = (amount - (0.5m * amount)) - disc * (amount - (0.5m * amount));
        }
        return result;
    }
}
```

4. Заданий код метода, який приймає рядок. Потім цей рядок шифрується та розшифровується. Потрібно визначити ознаки поганого стилю програмування та здійснити рефакторинг коду. Вставте в код документуючі коментарі.

```
public string security(string plainText)
{
    try
    {
        byte[] encrypted;
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream())
            using (CryptoStream cs = new CryptoStream(ms, encryptor,
                CryptoStreamMode.Write))
            {
                using (StreamWriter sw = new StreamWriter(cs))
                {
                    sw.Write(plainText);
                }
                encrypted = ms.ToArray();
            }
        }
        Console.WriteLine($"Encrypted data:
        { System.Text.Encoding.UTF8.GetString(encrypted)}");
        using (AesManaged aesm = new AesManaged())
        {
            ICryptoTransform decryptor = aesm.CreateDecryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream(encrypted))
            {
                using (CryptoStream cs = new CryptoStream(ms, decryptor,
                    CryptoStreamMode.Read))
                {
                    using (StreamReader reader = new StreamReader(cs))
                    {
                        plainText = reader.ReadToEnd();
                    }
                }
            }
        }
    }
}
```

```

        Console.WriteLine($"Decrypted data: {plainText}");
    }
    catch (Exception exp)
    {
        Console.WriteLine(exp.Message);
    }
    Console.ReadKey();
    return plainText;
}

```

5. Заданий клас Employee, який містить поля з даними про працівників та методи додавання даних в БД та друку звіту працівників. Який принцип SOLID порушений в цьому класі? Перепишіть код відповідно до принципів SOLID.

```

class Employee
{
    public int ID { get; set; }
    public string FullName { get; set; }
    ///<summary>
    ///метод додає в БД нового співробітника
    ///</summary>
    ///<paramname="emp">дані про співробітник, що додаються</param>
    ///<returns>Результат додавання нових даних</returns>
    public bool Add(Employee emp)
    {
        // Вставити дані про співробітника в таблицю БД
        return true;
    }
    ///<summary>
    ///Звіт співробітника
    ///</summary>
    public void GenerateReport(Employee emp)
    {
        Console.WriteLine("Генерація звіту з діяльності співробітника");
    }
}

class Program
{
    ///тестовий метод
    static void Main(string[] args)
    {
        Employee man = new Employee();
        man.GenerateReport(man);
    }
}

```

6. Проаналізуйте коди лабораторних робіт з дисципліни «Вступ до об'єктно-орієнтованого програмування» з точки зору структури класів, дотримання принципів SOLID, мінімізації і компактності класів, правил визначення імен, форматування та коментування коду. Перепишіть код з дотриманням вимог C# Coding Standard. Зафіксуйте свою роботу по рефакторингу коду у звіті до лабораторних робіт.

Використані джерела

1. Як структурне програмування допомагає створювати ефективний і надійний код. [Електронний ресурс]. – Режим доступу: <https://foxminded.ua/strukturne-prohramuvannia/>
2. Sutter H. and Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Pearson Education, Inc. 2005, 240 p.
3. 10 Essential Best Practices for Writing High-Quality C++ Source Code. *CppDepend*: веб-сайт. URL: <https://cppdepend.com/blog/master-cpp-top-best-coding-practices> (дата звернення 10.08.2024).
4. Useless коментарі до коду: які писати не варто і чому. *DOU*: веб-сайт. URL: <https://dou.ua/forums/topic/42553/> (дата звернення 10.08.2024).
5. C Code Style Guidelines. *cs.swarthmore.edu*: веб-сайт. URL: https://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html (дата звернення 10.08.2024).
6. Урок №12. Коментарі. *aCode*: веб-сайт. URL: <https://acode.com.ua/urok-12-komentari/> (дата звернення 10.08.2024).
7. Working Draft, Standard for Programming Language C++. *Open-std*: веб-сайт. – URL: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf> (дата звернення 10.08.2024).
8. Google C++ Style Guide. *Web.archive*: веб-сайт. URL: <https://web.archive.org/web/20150907011528/http://google-styleguide.googlecode.com/svn/trunk/cppguide.html> (дата звернення 10.08.2024).
9. C++ Core Guidelines. *isocpp*: веб-сайт. URL: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines> (дата звернення 10.08.2024).
10. Мартін Р. Чистий код. Створення, аналіз і рефакторинг. - Харків: "Фабула", 2019. - 416 с.
11. Gregoire M. Professional C++. Wrox; 5th edition. 2021. 1312 p.
12. Members (C# Programming Guide). *Microsoft Learn*: веб-сайт. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/members> (дата звернення 10.08.2024).
13. C# Coding Standards. *Dofactory*: веб-сайт. – URL: <https://www.dofactory.com/csharp-coding-standards> (дата звернення 10.08.2024).
14. C# Coding Standard. *SE-EDU*: веб-сайт. URL: <https://se-education.org/guides/conventions/csharp.html> (дата звернення 10.08.2024).
15. C# coding guideline. *Biss Handbook*: веб-сайт. URL: <https://handbook.bosphorusiss.com/coding-guideline/c-sharp-coding-guideline/> (дата звернення 10.08.2024).
16. C# Coding Standards for .NET. *State – NET*: веб-сайт. URL: <https://aspblogs.blob.core.windows.net/media/lhunt/Publications/CSharp%20Coding%20Standard s.pdf> (дата звернення 10.08.2024).