МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

ОБ’ЄКТНО-ОРІЄНТОВАНІ ТЕХНОЛОГІЇ ДЛЯ ЕЛЕКТРОННИХ ВИДАНЬ
ОПОРНИЙ КОНСПЕКТ ЛЕКЦІЙ

КИЇВ 2021

ЗМІСТ

ТЕМА 1. КЛАСИ І ОБ'ЄКТИ 4
Основні принципи об'єктно-орієнтованого програмування 4
Що таке клас в об'єктно-орієнтованому програмуванні 6
Загальна форма оголошення класу 8
Конструктори і деструктори 14
ТЕМА 2. МАСИВИ ОБ'ЄКТІВ, ВКАЗІВНИКИ І ПОСИЛАННЯ 25
Ініціалізація масиву значеннями, що є членами даними об'єкту 25
Ініціалізація масиву значеннями які повертають методи об'єкту класу 27
Масиви вказівників на методи, що є членами класу 28
Вказівник на статичний член даних класу 29
Використання членів даних класу 31
Оголошення та використання об'єкту класу, який є членом-даних іншого класу 33
Посилання на об'єкт класу 34
Види функцій-членів класу 37
Загальна форма оголошення функції з константним вказівником this 39
Загальна форма оголошення функції з непостійним вказівником this 42
ТЕМА 3. ВБУДОВАНІ ТА ДРУЖНІ ФУНКЦІЇ 45
inline функції-члени класу 45
Приклад оголошення класу, що містить вбудовані функції (inline) 46
Дружні класи та дружні функції 48
Приклад оголошення класу, що є дружнім до іншого класу 50
Приклад оголошення функції, що є дружньою до іншого класу 52
ТЕМА 4. ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ 56
Визначення терміну "перевантаження" функції 56
Приклади перевантаження функцій 57
Приклад перевантаження функції у класі 59
Умови перевантаження функції 60
Переваги перевантаження конструкторів класу 60
Доступ до перевантаженої функції з допомогою покажчика на функцію 62
Використання ключового слова overload 63
ТЕМА 5. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ 65
Унарні та бінарні оператори 65
Суть перевантаження операторів 65
Приклад перевантаження унарних та бінарних операторів для класу, який містить одиночні дані 66
Приклад перевантаження оператора '*', що обробляє клас, який містить масив дійсних чисел 69
Обмеження, які накладаються на перевантажені оператори 71
Перевантаження операторів +, –, *, / з допомогою "дружніх" операторних функцій 74

Приклад перевантаження бінарного оператора ‘–’ у класі для якого реалізована “дружня” операторна функція 75
ТЕМА 6. УСПАДКУВАННЯ....79
Види спадкування....79
Просте успадкування....80
Правила успадкування різних методів....84
Правила для деструкторів при успадкуванні....84
Множинне успадковування....85
Альтернатива успадковування....85
ТЕМА 7. ВІРТУАЛЬНІ ФУНКЦІЇ ТА ШАБЛОНИ....89
Віртуальні функції....89
Чиста віртуальна функція....92
Механізм пізнього зв’язування....92
Віртуальний деструктор....93
Абстрактні класи....95
Шаблони функцій....95
Шаблони класів....97
ТЕМА 8. СИСТЕМА ВВЕДЕННЯ-ВИВЕДЕННЯ....100
Базові положення системи введення-виведення C++....100
Форматне введення-виведення даних....101
Використання функцій width(), precision() і fill()....102
Маніпулятори введення-виведення....103
Файлове введення-виведення....103
Створення власних функцій вставки....105
Створення власних функцій вилучення....106
Створення власних маніпуляторів....106
СПИСОК ЛІТЕРАТУРИ....109

ТЕМА 1. КЛАСИ І ОБ'ЄКТИ

Зміст:

Основні принципи об'єктно-орієнтованого програмування
Що таке клас в об'єктно-орієнтованому програмуванні
Загальна форма оголошення класу
Конструктори і деструктори

Об'єктно-орієнтоване програмування - це методологія програмування, заснована на представленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного класу, а класи утворюють ієрархію спадкування.

Об'єкт - це щось, що має чітко визначені межі. Однак, цього недостатньо, щоб відокремити один об'єкт від іншого або дати оцінку якості абстракції. Об'єкт має стан, поведінку і ідентичність; структура і поведінка схожих об'єктів визначає загальний для них клас; терміни «екземпляр класу» і «об'єкт» взаємозамінні.

Клас - це безліч об'єктів, що володіють загальною структурою, поведінкою і семантикою. Окремий об'єкт - це екземпляр класу. Клас представляє лише абстракцію істотних властивостей об'єкта.

Стан об'єкта характеризується переліком (зазвичай статичним) усіх властивостей даного об'єкта та поточними (зазвичай динамічними) значеннями кожного з цих властивостей. Наприклад: торговий автомат має властивості: здатність приймати монети; цій властивості відповідає динамічне значення - кількість прийнятих монет.

Поведінка об'єкта - це те, як об'єкт діє і реагує; поведінка виражається в термінах стану об'єкта і передачі повідомлень. Операцією називається певним чином впливати одного об'єкта на інший з метою викликати відповідну реакцію.

Індивідуальність об'єкта - це така властивість об'єкта, яке відрізняє його від всіх інших об'єктів. У більшості мов програмування при створенні об'єкт називається, тому багато хто плутає адресується і індивідуальність. Неможливість відрізнити ім'я об'єкта від самого об'єкта є джерелом безлічі помилок в ООП.

Основні принципи об'єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування будується на трьох основних принципах: інкапсуляція, поліморфізм і успадкування.

Інкапсуляція

Інкапсуляція - це процес відділення один від одного елементів об'єкта, що визначають його пристрій і поведінку; інкапсуляція служить для того, щоб ізолювати контрактні зобов'язання абстракції від їх реалізації.

Нехай члену класу потрібен захист від «несанкціонованого доступу». Як розумно обмежити безліч функцій, яким такий член буде доступний? Очевидна відповідь для мов, що підтримують об'єктно-орієнтоване програмування, такий: доступ мають всі операції, які визначені для цього об'єкта, іншими словами, всі функції-члени. Наприклад:

```
class window
{
        // . . .
protected:
        Rectangle inside;
        // . . .
};

classs dumb_terminal: public window
{
        // . . .
public:
        void prompt ();
        // . . .
};
```

Тут в базовому класі window член inside типу Rectangle описується як захищений (protected), але функції-члени похідних класів, наприклад, dumb_ terminal :: prompt (), можуть звернутися до нього і з'ясувати, з якого виду вікном вони працюють. Для всіх інших функцій член window :: недоступний.

У такому підході поєднується високий ступінь захищеності з гнучкістю, необхідною для програм, які створюють класи і використовують їх.

Неочевидне наслідок з цього: не можна скласти повний і остаточний список всіх функцій, яким буде доступний захищений член, оскільки завжди можна додати ще одну, визначивши її як функцію-член в новому похідному класі. Для методу абстракції даних такий підхід часто буває мало прийнятним. Якщо мова орієнтується на метод абстракції даних, то очевидне для нього рішення - це вимога вказувати в описі класу список всіх функцій, яким потрібен доступ до члена. В C ++ для цієї мети використовується опис приватних (private) членів.

Важливість інкапсуляції, тобто укладення членів в захисну оболонку, різко зростає зі зростанням обсягів програми і збільшується розкидом областей додатку.

Спадкування

Спадкування є здатність виробляти новий клас з існуючого базового класу. Похідний клас - це новий клас, а базовий клас - існуючий клас. Коли ви породжує один клас з іншого (базового класу), похідний клас успадковує елементи базового класу. Для породження класу з базового починайте визначення похідного класу ключовим словом class, за яким слідує ім'я класу, двокрапка і ім'я базового класу, наприклад class dalmatian: dog.

Коли ви породжує клас з базового класу, похідний клас може звертатися до загальних елементів базового класу, як ніби ці елементи визначені всередині самого похідного класу. Для доступу до приватних даними базового класу похідний клас повинен використовувати інтерфейсні функції базового класу.

Усередині конструктора похідного класу ваша програма повинна викликати конструктор базового класу, вказуючи двокрапка, ім'я конструктора базового

класу і відповідні параметри відразу ж після заголовка конструктора похідного класу.

Щоб забезпечити похідним класам прямий доступ до певних елементів базового класу, в той же час захищаючи ці елементи від решти програми, C ++ забезпечує захищені (protected) елементи класу. Похідний клас може звертатися до захищених елементів базового класу, як ніби вони є загальними. Однак для решти програми захищені елементи еквівалентні приватним.

Якщо в похідному і базовому класі є елементи з однаковим ім'ям, то всередині функцій похідного класу C ++ буде використовувати елементи похідного класу. Якщо функцій похідного класу необхідно звернутися до елементу базового класу, ви повинні використовувати оператор глобального дозволу, наприклад base class :: member.

Поліморфізм

Поліморфний об'єкт являє собою такий об'єкт, який може змінювати форму під час виконання програми.

В об'єктно-орієнтованих мовах клас є абстрактним типом даних. Поліморфізм реалізується за допомогою успадкування класів і віртуальних функцій. Клас-нащадок успадковує сигнатури методів класу-батька, а реалізація, в результаті перевизначення методу, цих методів може бути інший, що відповідає специфіці класу-нащадка. Інші функції можуть працювати з об'єктом класом-батьком, але при цьому замість нього під час виконання буде підставлятися один з класів-нащадків. Це називається пізнім зв'язуванням.

Клас-нащадок сам може бути батьком. Це дозволяє будувати складні схеми спадкування - деревовидні або мережні.

Абстрактні (або чисто віртуальні) методи не мають реалізації взагалі. Вони спеціально призначені для наслідування. Їх реалізація повинна бути визначена в класах-нащадках.

Клас може успадковувати функціональність від декількох класів. Це називається множинним спадкуванням. Множинне успадкування створює проблему (коли клас успадковується від декількох класів-посередників, які в свою чергу успадковуються від одного класу (так звана «Проблема ромба»): якщо метод загального предка був перевизначений в посередниках, невідомо, яку реалізацію методу повинен наслідувати загальний нащадок. вирішується ця проблема через віртуальне успадкування.

Що таке клас в об'єктно-орієнтованому програмуванні

В остаточному вигляді будь-яка програма являє собою набір інструкцій процесора. Все, що написано на будь-якій мові програмування - більш зручна, спрощена запис цього набору інструкцій, що полегшує написання, налагодження і подальшу модифікацію програми. Чим вище рівень мови, тим в більш простій формі записуються одні і ті ж дії.

Зі збільшенням обсягу програми стає неможливим утримувати в пам'яті

всі деталі, і стає необхідним структурувати інформацію, виділяти головне й відкидати несуттєве. Цей процес називається підвищенням ступеня абстракції програми.

Для мови високого рівня першим кроком до підвищення абстракції є використання функцій, що дозволяє після написання і налагодження функції відволіктися від деталей її реалізації, оскільки для виклику функції потрібно знати тільки її інтерфейс. Якщо глобальні змінні не використовуються, інтерфейс повністю визначається заголовком функції.

Наступний крок - опис власних типів даних, що дозволяють структурувати і групувати інформацію, представляючи її в більш природному вигляді. Наприклад, всі різнорідні відомості, що відносяться до одного виду товару на складі, можна уявити за допомогою однієї структури.

Для роботи з власними типами даних необхідні спеціальні. Природно згрупувати їх з описом цих типів даних в одному місці програми, а також по можливості відокремити від її інших частин. При цьому для використання цих типів і функцій не потрібно повного знання того, як саме вони написані - необхідні тільки опису інтерфейсів. Об'єднання в модулі описів типів даних і функцій, призначених для роботи з ними, з приховуванням від користувача модуля несуттєвих деталей є подальшим розвитком структуризації програми.

Всі три описаних вище методу підвищення абстракції мають на меті спростити структуру програми, тобто уявити її у вигляді меншої кількості більших блоків і мінімізувати зв'язку між ними. Це дозволяє управляти великим обсягом інформації і, отже, успішно налагоджувати більш складні програми.

Введення поняття класу є природним розвитком ідей модульності. У класі структури даних і функції їх обробки об'єднуються. Клас використовується тільки через його інтерфейс - деталі реалізації для користувача класу не істотні.

Ідея класів відображає будову об'єктів реального світу - адже кожен предмет або процес має набір характеристик або відмінних рис, іншими словами, властивостями і поведінкою. Програми в основному призначені для моделювання предметів, процесів і явищ реального світу, тому зручно мати в мові програмування адекватний інструмент для представлення моделей.

Клас є типом даних, що визначаються користувачем. У класі задаються властивості і поведінку будь-якого предмета або процесу у вигляді полів даних (аналогічно структурі) і функцій для роботи з ними. Створюваний тип даних має практично тими ж властивостями, що і стандартні типи (нагадаю, що тип задає внутрішнє представлення даних в пам'яті комп'ютера, безліч значень, яке можуть приймати величини цього типу, а також операції і функції, що застосовуються до цих величин).

Істотною властивістю класу є те, що деталі його реалізації приховані від користувачів класу за інтерфейсом. Інтерфейсом класу є заголовки його відкритих методів. Таким чином, клас як модель об'єкта реального світу є чорним ящиком, замкнутим по відношенню до зовнішнього світу.

Ідея класів є основою об'єктно-орієнтованого програмування (ООП).

Основні принципи ООП були розроблені ще в мовах Simula-67 і Smalltalk, але в той час не отримали широкого застосування через труднощі освоєння і низьку ефективність реалізації. В C++ ці концепції реалізовані ефективно і несуперечливо, що і стало основою успішного поширення цієї мови і впровадження подібних засобів в інші мови програмування.

Ідеї ООП не надто прості для практичного використання (їх неграмотне застосування приносить набагато більше шкоди, ніж користі), а освоєння існуючих стандартних бібліотек вимагає часу і високого рівня початкової підготовки.

Конкретні змінні типу даних «клас» називаються екземплярами класу, або об'єктами. Об'єкти взаємодіють між собою, посилаючи і отримуючи повідомлення. Повідомлення - це запит на виконання дії, що містить набір необхідних параметрів. Механізм повідомлень реалізується за допомогою виклику відповідних функцій. Таким чином, за допомогою ООП легко реалізується так звана «подієво-керована модель», коли дані активні і управляють викликом того чи іншого фрагмента програмного коду.

Прикладом реалізації подієво-керованої моделі може служити будь-яка програма, керована за допомогою меню. Після запуску така програма пасивно очікує дій користувача і повинна вміти правильно відреагувати на будь-який з них. Подієва модель є протилежністю традиційної (директивної), коли код управляє даними: програма після старту пропонує користувачеві виконати деякі дії (ввести дані, вибрати режим) відповідно до жорстко заданим алгоритмом.

Загальна форма оголошення класу

Клас визначає формат (опис) деяких даних та роботу (поведінку) над цими даними. З оголошення класу можна отримати різну кількість об'єктів класу (змінних типу "клас"). Кожен об'єкт класу визначається конкретним (на даний момент) значенням внутрішніх даних, який називають станом об'єкту.

У класі оголошуються дані (внутрішні змінні, властивості) та методи (функції), що оперують цими даними (виконують роботу над даними).

У середовищі CLR підтримуються два види класів:

- некеровані (unmanaged) класи. Для виділення пам'яті під об'єкти таких класів можуть бути використані некеровані покажчики (*) та операція new;
- керовані (managed) класи. Для виділення пам'яті в таких класах можуть бути використані керовані покажчики (^) та операція gcnew.

Дана тема висвітлює особливості використання некерованих (*) класів.

У найпростішому випадку (без спадковості) загальна форма оголошення unmanaged-класу має такий вигляд

```
class імя_класу
{
  private:
  // приховані дані та методи (функції)
  // ...
```

```
    public:
    // відкриті дані та методи
    // ...

    protected:
    // захищені дані та методи
    // ...
};
```

де ім'я_класу – безпосередньо ім'я типу даних "клас". Це ім'я використовується при створенні об'єктів класу.

Ключове слово class повідомляє про те, що оголошується новий клас (тип даних). Всередині класу оголошуються члени класу: дані та методи. Ключове слово private визначає члени класу, що мають бути приховані (закриті) від зовнішніх методів, оголошених за межами класу, а також об'єктів класу. Члени даних, оголошені з ключовим словом private, доступні тільки іншим членам цього класу.

Ключове слово public визначає дані (змінні) та методи (функції) класу, що є загальнодоступними.

Ключове слово protected визначає захищені дані та методи класу, які є:

- доступні для методів успадкованих класів, якщо даний клас є батьківським по відношенню до інших класів;
- недоступні для інших методів, що реалізовані в інших частинах програми;
- недоступні для об'єктів (екземплярів) класу.

В межах опису класу секції (розділи) private, protected, public можуть слідувати в будь-якому порядку та в будь-якій кількості.

Наприклад:

```
class MyClass
{
    // секція (розділ) private за замовчуванням
    // ...

    public:
    // секція public
    // ...

    protected:
    // секція protected
    // ...

    private:
```

```
    // знову секція private
    // ...

    public:
    // знову секція public
    // ...
};
```

Термін «інкапсуляція даних» означає, що для членів класу (даних та методів) можна встановлювати ступінь доступності з інших частин програмного коду (інших методів, об'єктів класу). Таким чином, виникає поняття приховування даних (методів) у класі.

Інкапсуляція забезпечує покращення надійності зберігання даних в класі шляхом вводу додаткових методів перевірки цих даних на допустимі значення. Як правило, доступ до прихованих даних в класі відбувається не напряму, а через виклики спеціальних методів доступу чи властивостей. Безпосередньо дані розміщуються у прихованій частині класу, а методи доступу до цих даних розміщуються у загальнодоступній частині класу.

Класична мова C++ дозволяє встановлювати доступ до членів класу з допомогою трьох специфікаторів: private, protected, public.

private – члени класу є прихованими. Це означає, що доступ до них мають тільки методи, що оголошені в класі. private-члени класу є недоступними з породжених класів та об'єктів цього класу;

protected – члени класу є захищеними. Це означає, що доступ до protected-членів мають методи класу, "дружні функції" та методи успадкованих класів. protected-члени класу є недоступними для об'єктів цього класу;

public – члени класу є відкритими (доступними) для усіх методів та об'єктів з усіх інших частин програмного коду.

Клас може бути оголошений без методів. Такі класи містять тільки дані. Щоб доступитися до даних у класі що не містить методів, потрібно ці дані оголосити в розділі public. Класи без методів майже не застосовуються. Якщо оголосити дані в розділі private, то доступитись до членів-даних класу буде неможливо.

Приклад. У даному прикладі оголошується клас без методів, що реалізує операції над датою. Клас містить внутрішні змінні (дані), що представляють собою число, місяць, рік.

```
class CMyDate
{
    public:
    int day; // число
    int month; // місяць
    int year; // рік
};
```

Фрагмент коду, що демонструє роботу з класом CMyDate

```
// оголосити об'єкт класу MyDate
CMyDate D;

// заповнити значеннями поля класу
D.day = 20;
D.month = 02;
D.year = 2002;

int t = D.year; // t = 2002
t = D.month; // t = 2
```

Клас може бути без даних та без методів. Наприклад, нижче оголошено клас, що немає ні даних ні методів.

```
// клас не має ні даних ні методів
class NoData
{

};
```

Об'єкт такого класу також створюється.

```
NoData nd; // об'єкт класу NoData
```

Зрозуміло, що такий клас має дуже обмежене використання. Пустий клас доцільно створювати у випадку, коли під час створення великого проекту потрібно протестувати його ранню (початкову) версію, в якій деякі класи ще не розроблені і не реалізовані. Замість реального класу вказується пустий клас – заглушка. Цей клас розроблений під потреби майбутнього проекту таким чином, щоб компілятор не видавав повідомлення про помилку і можна було протестувати вже написану частину коду. У цьому випадку ім'я пустого класу вибирається таким, яким воно повинно бути в майбутньому проекті.

Основні переваги класів виявляються при наявності методів – членів класу. З допомогою методів доступу до даних в класах можна зручно:

- ініціалізувати дані початковими значеннями;
- здійснювати перевірку на коректність внутрішніх даних при їх задаванні;
- реалізовувати різні варіанти перетворення (конвертування) даних;
- реалізовувати організацію виділення пам'яті для складених типів даних, що реалізовані у класах;
- виконувати різноманітні види обчислень над даними.

Приклад. Модифікація класу CMyDate. Клас, що описує дату та операції над нею. Операції доступу до членів класу реалізовані з допомогою відповідних методів. Самі дані реалізовані у розділі private.

Програмний код класу

```
class CMyDate
{
   int day;
   int month;
   int year;

   public:
   void SetDate(int d, int m, int y); //
   int GetDay(void); // повертає номер дня
   int GetMonth(void); // повертає номер місяця
   int GetYear(void); // повертає
};
```

Реалізація методів класу SetDate(), GetDay(), GetMonth(), GetYear()

```
// Встановити нову дату
void CMyDate::SetDate(int d, int m, int y)
{
   day = d;
   month = m;
   year = y;
}

// зчитати номер дня
int CMyDate::GetDay(void)
{
   return day;
}

// зчитати номер місяця
int CMyDate::GetMonth(void)
{
   return month;
}

// зчитати рік
int CMyDate::GetYear(void)
{
```

```
    return year;
}
```

Програмний код методів-членів класу можна описувати в самому класі та за його межами. Якщо потрібно описати код методу, що є членом класу, то для цього використовується оператор розширення області видимості "::" . Оператор "::" визначає ім'я члену класу разом з іменем класу, в якому він реалізований.

Оголошення класу – це опис формату типу даних. Цей тип даних "клас" описує дані та методи, що оперують цими даними. Опис класу – це тільки опис (оголошення). У цьому випадку пам'ять для класу не виділяється.

Пам'ять виділяється тільки тоді, коли клас використовується для створення об'єкту. Цей процес ще називають створенням екземпляру класу, що являє собою фізичну сутність класу.

Оголошення об'єкту класу (екземпляру) нічим не відрізняється від оголошення змінної:

```
Ім'я_класу ім'я_об'єкту;
```

З допомогою імені ім'я_об'єкту можна здійснити доступ до загальнодоступних (public) членів класу. Це здійснюється з допомогою символу ' . ' (крапка).

Можливий варіант оголошення покажчика на клас. Якщо це unmanaged-клас, то оголошення має вигляд:

```
Ім'я_класу * ім'я_об'єкту;
```

Після такого оголошення, потрібно виділяти пам'ять для об'єкту класу з допомогою оператора new. Доступ до даних за покажчиком здійснюється з допомогою комбінації символів '->' так само як і у випадку зі структурами.

Наприклад. Оголошення класу Worker, що описує методи та дані про працівника підприємства.

```
class Worker
{
   private:
   // private-члени класу Worker
   // ...

   public:
   // public-члени класу Worker
   // ...
};
```

Об'єкт класу – це змінна типу "клас". При оголошенні об'єкту класу виділяється пам'ять для цього об'єкту (змінної). Наприклад, для класу Worker можна написати наступний код

```
Worker w1; // об'єкт класу Worker, виділяється пам'ять для даних об'єкту
```

З об'єкту можна мати доступ тільки до public-членів класу. Це можна здійснювати з допомогою символу ' . '(крапка) або доступу за покажчиком '->';

```
w1.SetName(«Johnson J.»); // виклик public-методу
Worker * w2; // покажчик на клас Worker
w2 = new Worker(); // динамічне виділення пам'яті для класу
w2->SetName(«Jackson M.»); // виклик public-методу з класу за покажчиком
```

За замовчуванням, члени класу мають доступ private. Тому, при оголошенні класу, якщо потрібно вказати private-члени класу, це слово можна опустити.

Як правило, private-члени класу є закритими. Це є основна перевага інкапсуляції. Щоб змінювати значення private-членів класу, використовують методи класу, що оголошені в public-секції. У цих методах можна змінювати значення private-членів. Такий підхід використовується для забезпечення надійності збереження даних у private-членах. У public-методах, що мають доступ до private-членів, можна реалізувати додаткові перевірки на допустимість значень.

Конструктори і деструктори

Клас може містити спеціальні функції: конструктори і деструктори.

Конструктор класу – це спеціальний метод (функція) класу. Конструктор викликається при створенні об'єкту класу. Як правило, конструктор використовується для:

- виділення пам'яті для об'єкту класу;
- початкової ініціалізації внутрішніх даних класу.

Конструктор призначений для формування екземпляру об'єкту класу.

Ім'я конструктора класу співпадає з іменем класу.

Виклик конструктора здійснюється при створенні об'єкту класу. Конструктор класу викликається компілятором.

Конструктор може мати будь-яку кількість параметрів. Також конструктор може бути без параметрів (конструктор за замовчуванням).

Приклад. Оголошується клас CMyDate, що визначає дату (число, місяць, рік). У класі оголошено два конструктори. Один конструктор без параметрів, інший конструктор, що отримує три параметри, що встановлюють нову дату.

Оголошення класу та його методів має вигляд

```
// клас, що визначає дату
class CMyDate
{
   int day;
   int month;
   int year;

   public:
   // конструктори класу
   CMyDate(); // конструктор без параметрів
   CMyDate(int d, int m, int y); // конструктор з трьома параметрами

   // методи класу
   void SetDate(int d, int m, int y); // встановити нову дату
   int GetDay(void); // повертає номер дня
   int GetMonth(void); // повертає номер місяця
   int GetYear(void); // повертає рік
};

// реалізація конструкторів та методів класу
// конструктор без параметрів (конструктор за замовчуванням)
CMyDate::CMyDate()
{
   // встановити дату 01.01.2001
   day = 1;
   month = 1;
   year = 2001;
}

// конструктор з трьома параметрами
CMyDate::CMyDate(int d, int m, int y)
{
   day = d;
   month = m;
   year = y;
}

// Встановити нову дату
void CMyDate::SetDate(int d, int m, int y)
{
   day = d;
   month = m;
   year = y;
```

```
}

// зчитати номер дня
int CMyDate::GetDay(void)
{
   return day;
}

// зчитати номер місяця
int CMyDate::GetMonth(void)
{
   return month;
}

// зчитати рік
int CMyDate::GetYear(void)
{
   return year;
}
```

Демонстрація виклику конструкторів при оголошенні об'єктів класу

```
CMyDate MD1; // викликається конструктор без параметрів
CMyDate MD2(4, 5, 2008); // викликається конструктор з трьома параметрами

int t;
t = MD1.GetDay(); // t = 1
t = MD1.GetYear(); // t = 2001

t = MD2.GetMonth(); // t = 5
t = MD2.GetYear(); // t = 2008
```

Не обов'язково в класі описувати конструктор. При створенні об'єкту класу, що не містить жодного конструктора, буде викликатись так званий конструктор за замовчуванням (default constructor).

Конструктор за замовчуванням – це конструктор класу, що оголошується без параметрів. Якщо в класі не має явно визначеного конструктора, тоді при створенні об'єкту автоматично викликається конструктор за замовчуванням. Конструктор за замовчуванням просто виділяє пам'ять для об'єкту класу, коли він оголошується.

Приклад 1. Нехай задано клас CMyPoint, що визначає точку на координатній площині. У класі не реалізовано жодного конструктора.

```
// клас, що визначає точку на координатній площині
class CMyPoint
{
   int x;
   int y;

   public:
   // методи класу
   void SetPoint(int nx, int ny)
   {
      x = nx;
      y = ny;
   }

   int GetX(void) { return x; }
   int GetY(void) { return y; }
};
```

Однак, при створенні об'єкту класу компілятор автоматично викликає конструктор за замовчуванням.

```
CMyPoint MP; // автоматично викликається конструктор за замовчуванням
MP.SetXY(4, -10); // виклик методів класу

int t;
t = MP.GetY(); // t = -10
```

Конструктор за замовчуванням автоматично викликається тільки тоді, коли в класі не оголошено жодного конструктора. Як тільки в класі оголосити будь-який інший конструктор з параметрами, то при оголошенні

```
CMyPoint MP;
```

компілятор видасть помилку.

Приклад 2. Модифікація класу CMyPoint. У класі явно задано конструктор за замовчуванням.

```
// клас, що визначає точку на координатній площині
class CMyPoint
{
   int x;
   int y;
```

```
    public:
    // явно заданий конструктор за замовчуванням
    CMyPoint()
    {
       x = y = 0;
    }

    // методи класу
    void SetPoint(int nx, int ny)
    {
       x = nx;
       y = ny;
    }

    int GetX(void) { return x; }
    int GetY(void) { return y; }
};
```

Демонстрація виклику явно заданого конструктора за замовчуванням

```
CMyPoint MP; // викликається явно заданий конструктор за замовчуванням

int t;
t = MP.GetX(); // t = 0
```

Кожен клас може мати тільки один конструктор за замовчуванням. Це зв'язано з тим, що у класі не може бути двох методів (функцій) з однаковою сигнатурою.

Конструктор не може повертати значення (навіть значення void). Якщо в конструкторі написати повернення значення з допомогою оператора return, то компілятор видасть помилку.

Якщо в класі оголошено об'єкт іншого класу (підоб'єкт), то в цьому випадку:

- першим викликається конструктори (конструктори) класу, що є підоб'єктом включаючого класу;
- наступним викликається конструктор включаючого класу.

Якщо є два класи, один з яких базовий а інший – породжений (успадкований) від базового, то в цьому випадку послідовність викликів наступна:

- спочатку викликається конструктор базового класу;
- наступним викликається конструктор породженого класу.

Конструктор може оголошуватись у розділі private.

Такий конструктор називається приватним конструктором. Приватний конструктор викликається з функцій даного класу та функцій, які є "дружніми" до даного класу.

При оголошенні звичайного об'єкту класу, конструктори, які розміщені в

розділі private (приватні конструктори), є недоступними.

Щоб використовувати приватні конструктори, потрібно виконання однієї з трьох умов:

- конструктор викликається статичним членом класу;
- конструктор викликається дружнім класом;
- у класі оголошена функція-член, яка викликає даний конструктор для створення нового об'єкту.

В класі може бути оголошено два конструктори, що приймають однакову кількість параметрів. Однак з умовою, що типи параметрів будуть відрізнятись. Для класу повинно виконуватись правило: у класі не може бути двох методів (функцій) з однаковою (співпадаючою) сигнатурою.

Параметризований конструктор – це конструктор класу, що має параметри.

Способи ініціалізації членів об'єкту з допомогою конструктора, що отримує один параметр:

- ініціалізація за зразком виклику функції;
- ініціалізація за зразком оператора присвоєння.

Конструктор копіювання – це конструктор, який дозволяє отримати ідентичний до заданого об'єкт. Тобто, з допомогою конструктора копіювання можна отримати копію вже існуючого об'єкту. Конструктор копіювання ще називається ініціалізатором копії (copy initializer). Конструктор копіювання повинен отримувати вхідним параметром константне посилання (&) на об'єкт такого самого класу.

Конструктор копіювання викликається у випадках, коли потрібно отримати повну копію об'єкту. В C++ необхідність отримання повної копії об'єкту можлива у трьох випадках.

Випадок 1. При оголошенні нового об'єкту та його ініціалізації даними іншого об'єкту з допомогою оператора=. Наступний фрагмент коду демонструє дану ситуацію для деякого класу ClassName

```
// оголошення екземпляру (об'єкту) obj1 класу ClassName
ClassName obj1;

// оголошення об'єкту obj2 з одночасною ініціалізацією даними об'єкту obj1
ClassName obj2 = obj1; // викликається конструктор копіювання
```

У цьому випадку потрібно скопіювати дані з об'єкта obj1 в об'єкт obj2. Тобто, потрібно створити копію об'єктуobj1 так, щоб цей об'єкт obj1 міг в подальшому коректно використовуватись у програмі. Тому, необхідна копія. Цим займається конструктор копіювання.

Випадок 2. Коли потрібно передати об'єкт у функцію як параметр-значення. У цьому випадку створюється повна копія об'єкту.

Випадок 3. Коли потрібно повернути об'єкт з функції за значенням. У цьому випадку також створюється повна копія об'єкту, що повертається.

Конструктор копіювання необхідно використовувати у тих класах, в яких здійснюється динамічне виділення пам'яті для даних. Іншими словами, якщо у класі є покажчик, для даних якого пам'ять виділяється динамічно з допомогою оператора new (або іншими функціями), то такий клас обов'язково повинен містити конструктор копіювання. Інакше, у програмі будуть виникати проблеми, що пов'язані з недоліками побітового копіювання. Крім того, такий клас ще повинен містити операторну функцію operator=(), яка перевантажує оператор копіювання (це вже інша тема).

Якщо у класі немає динамічного виділення пам'яті для даних, то конструктор копіювання можна не використовувати. У цьому випадку побітового копіювання (за замовчуванням) цілком достатньо для коректної роботи класу. Виняток, якщо при ініціалізації об'єкта іншим об'єктом потрібно встановити деякі спеціальні умови копіювання.

Якщо у класі не оголошений конструктор копіювання, то використовується конструктор копіювання, який автоматично генерується компілятором. Цей конструктор копіювання здійснює побітове копіювання для отримання копії об'єкта.

Побітове копіювання є коректним для класів, в яких немає динамічного виділення пам'яті. Однак, якщо у класі є динамічне виділення пам'яті (клас використовує покажчики), то побітове копіювання призводить до того, що покажчики двох об'єктів вказують на ту саму ділянку пам'яті. А це є помилка.

Деструктор, як і конструктор, відноситься до спеціальних функцій класу. Деструктор – це спеціальний метод, що викликається при видаленні об'єкту. Як правило, деструктор використовується для звільнення пам'яті, динамічно виділеної під внутрішні дані класу. Можуть бути й інші випадки застосування деструктора.

Деструктор – це зворотна по відношенню до конструктора функція.

Ім'я деструктора співпадає з іменем класу, перед яким слідує символ '~'. Наприклад, якщо клас має ім'яCMyClass, тоді ім'я деструктора буде ~CMyClass().

Приклад використання загальнодоступного деструктора. Нехай задано клас, що визначає масив структур типу DATE. Кожна структура в масиві описує дату в форматі: число, місяць, рік. У класі оголошено ряд конструкторів, методів а також деструктор, який звільняє пам'ять для масиву.

Загальний код модуля, в якому оголошується структуру та клас.

```
// структура, що реалізує дату
struct DATE
{
   int day;
   int month;
   int year;
```

```
};

// клас, що реалізує масив структур типу DATE
class CDates
{
   int n;
   DATE *A; // масив структур

   public:
   CDates(void); // конструктор за замовчуванням
   ~CDates(void); // деструктор

   // методи класу
    void SetN(int nn); // встановити нове значення n, перевиділити пам'ять для масиву A
    DATE GetDATE(int index); // повернути значення структури з індексом index
};

// конструктор за замовчуванням
CDates::CDates(void)
{
   n = 0;
}

// деструктор
CDates::~CDates(void)
{
   // звільнення пам'яті, виділеної для масиву структур
   delete A;
}

// встановити нове значення n
void CDates::SetN(int nn)
{
   // 1. Звільнити попередню пам'ять
   // звільнити пам'ять, виділену під масив структур
   delete A;

   // 2. Встановити нове n
   n = nn;

   // 3. Виділити пам'ять для масиву покажчиків на структуру типу (DATE
```

```
*)
        A = new DATE[n];

        // заповнити значення кожної структури довільними значенням
        for (int i=0; i<n; i++)
        {
           A[i].day = i;
           A[i].month = i%12 + 1;
           A[i].year = 1000*i;
        }
     }

     // повернути значення структури з індексом DATE
     DATE CDates::GetDATE(int index)
     {
        return (DATE)A[index];
     }
```

Демонстрація використання даного класу в іншому методі.

```
     CDates CD; // оголосити об'єкт типу CDates, викликається конструктор за замовчуванням

     // встановити нове значення n
     CD.SetN(8);

     // перевірка
     int t;
     DATE D;
     D = CD.GetDATE(5); // взяти структуру з індексом 5

     t = D.day; // t = 5
     t = D.year; // t = 5000

     // після виходу з методу, об'єкт CD знищується, а, отже, викликається деструктор ~CDates
     // який звільняє пам'ять для масиву A в класі
```

Деструктор можна оголосити в розділі private. Такий деструктор називається приватним деструктором.

Використання приватних деструкторів доцільно у тих випадках, коли звичайним користувачам забороняється звільняти пам'ять для раніше створених об'єктів (знищувати раніше створені об'єкти).

Якщо в класі оголошено приватний деструктор, то виникають наступні обмеження:

- неможливо створити автоматичний об'єкт класу (об'єкт класу, що оголошується в деякому методі);
- неможливо створити статичний об'єкт;
- неможливо створити глобальний об'єкт класу.

Це зв'язано з тим, що такі об'єкти в подальшому неможливо буде знищити.

Наприклад. Нехай задано клас, в якому оголошено приватний деструктор.

```
class CMyInt
{
   private:
   int n; // кількість елементів масиву

   // деструктор, оголошений в розділі private
   ~CMyInt(void);

   public:
   CMyInt(void);

   // методи класу
   void SetN(int nn) { n = nn; }
   int GetN(void) { return n; }
};
```

Якщо спробувати створити об'єкт класу в іншому програмному коді на зразок

```
CMyInt MI;
```

то компілятор видасть помилку

'CMyInt::~MyInt': cannot access private member declared in class 'CMyInt'

Однак, можна оголошувати покажчик на клас CMyInt. Наступний опис буде добрим

```
// приватний деструктор
CMyInt * pMI = new CMyInt; // можна оголошувати покажчик на клас, що визначає приватний деструктор
```

Деструктор не може мати параметрів.

При використанні в класі, між конструктором і деструктором можна визначити такі основні відмінності:

- деструкторам не можна передавати параметри, конструкторам можна;
- деструктори можуть бути віртуальні, конструктори – ні;
- при оголошенні класу, можна оголосити тільки один деструктор. Однак, конструкторів можна оголошувати скільки завгодно. Головне, щоб вони відрізнялись сигнатурою.

Контрольні завдання та запитання

1. Наведіть визначення класу.
2. Наведіть визначення об’єкта.
3. Назвіть принципи ООП.
4. Що таке інкапсуляція?
5. Яка відмінність між закритими та відкритими членами класу?
6. Напишіть синтаксис описання класу.
7. Який синтаксис описання функції-члена класу?
8. Як оголошуються об’єкти? Яка відмінність між класом та об’єктом?
9. Яким чином організовано доступ до членів класу?
10. Нехай маємо клас widgit. Які імена його конструктора та деструктора?

```
class widgit {  int x, y;
public: /*...впишіть конструктори та деструктори*/
};
```

ТЕМА 2. МАСИВИ ОБ'ЄКТІВ, ВКАЗІВНИКИ І ПОСИЛАННЯ

Зміст:

Ініціалізація масиву значеннями, що є членами даними об'єкту
Ініціалізація масиву значеннями які повертають методи об'єкту класу
Масиви вказівників на методи, що є членами класу
Вказівник на статичний член даних класу
Використання членів даних класу
Посилання на об'єкт класу
Види функцій-членів класу
Загальна форма оголошення функції з константним вказівником this
Загальна форма оголошення функції з непостійним вказівником this

Ініціалізація масиву значеннями, що є членами даними об'єкту

Можливі випадки, коли потрібно об'єднати змінні, які є членами об'єкта класу в масив. У нижченаведеному прикладі ініціалізується масив, що в якості ініціалізаторів використовує члени-даних об'єкту. На момент оголошення члени-даних об'єкту мають мати якісь змістовні значення.

Приклад визначає клас Radius, в якому оголошуються:

- внутрішня private змінна radius;
- внутрішній private метод CalcValues(), що заповнює значеннями public змінні length, area, volume;
- загальнодоступні (public) внутрішні змінні класу length, area, volume, які містять відповідні значення довжини кола, площі круга та об'єму кулі для радіуса r;
- конструктори класу;
- методи доступу GetR(), SetR().

Лістинг класу наступний:

```
// клас, що реалізує величину радіуса деякої геометричної фігури
class Radius
{
   double radius; // змінна, що визначає радіус

   // внутрішній метод, що формує значення змінних length, area, volume
   void CalcValues(void)
   {
      const double pi = 3.141592653589;
      length = 2 * pi * radius;
      area = pi * radius * radius;
      volume = 4.0/3.0 * pi * radius * radius * radius;
   }

   public:
```

```
    double length; // довжина кола
    double area; // площа круга
    double volume; // об'єм кулі

    // конструктори
    Radius()
    {
       radius = 1;
       CalcValues(); // заповнити змінні length, area, volume
    }
    Radius(double radius)
    {
       this->radius = radius;
       CalcValues(); // заповнити змінні length, area, volume
    }

    // методи доступу
    int GetR(void) { return radius; }
    void SetR(double radius)
    {
       this->radius = radius;
       CalcValues(); // заповнити змінні length, area, volume
    }
};
```

Використання класу у функції main() або іншій функції:

```
Radius r(3); // об'єкт r класу Radius

// ініціалізація масиву V значеннями, що є членами даних об'єкту r
double V[] = {
   r.length,
   r.area,
   r.volume
};

double x;
x = V[0]; // x = 18.8496 - довжина кола
x = V[1]; // x = 28.2743 - площа круга
x = V[2]; // x = 113.097 - об'єм кулі
```

Ініціалізація масиву значеннями які повертають методи об'єкту класу

Елементи масиву можуть бути ініціалізовані значеннями, які повертають методи деякого класу. У нижченаведеному прикладі демонструється ініціалізація масиву V значеннями, які повертають public методи класу Radius: Length(), Area(), Volume().

Оголошення класу має вигляд:

```
// клас, що реалізує величину радіуса деякої геометричної фігури
class Radius
{
   double radius; // змінна, що визначає радіус

   public:
   double Length()
   {
      return (2 * 3.1415 * radius);
   }

   double Area()
   {
      return (3.1415 * radius * radius);
   }

   double Volume()
   {
      return (4.0/3.0 * 3.1415 * radius * radius * radius);
   }

   // конструктори
   Radius()
   {
      radius = 1;
   }
   Radius(double radius)
   {
      this->radius = radius;
   }

   // методи доступу
   int GetR(void) { return radius; }
   void SetR(double radius)
   {
```

```
        this->radius = radius;
    }
};
```

Використання класу в деякому методі

```
Radius r(3); // об'єкт r класу Radius

// ініціалізація масиву V значеннями, які повертають методи об'єкту класу r
double V[] = {
    r.Length(),
    r.Area(),
    r.Volume()
};

double x;
x = V[0]; // x = 18.8496 - довжина кола
x = V[1]; // x = 28.2743 - площа круга
x = V[2]; // x = 113.097 - об'єм кулі
```

Масиви вказівників на методи, що є членами класу

Дозволяється оголошувати вказівник на метод деякого класу. Отже, можна оголошувати масив вказівників на методи деякого класу. Зрозуміло, що сигнатура методів має бути однаковою. Щоб оголосити масив вказівників на методи класу A потрібно оголосити тип, що є вказівником на метод класу A. Для цього потрібно використати засіб typedef за зразком, як показано нижче

```
typedef type (A:: * PType)(arguments_list);
```

де type – тип, що повертають методи класу;

argument_list – список аргументів методів класу (має бути однаковий для усіх методів);

A – назва класу, в якому оголошуються методи, які потрібно об'єднати в масив;

PType – назва нового типу, який визначає вказівник на метод з класу A.

Якщо визначено тип PType, то масив вказівників можна оголосити за такою формою

```
PType ArrayName[size];
```

де PType – наперед визначений тип, що є вказівником на метод класу A;

ArrayName – ім'я масиву вказівників на методи класу A;

size – кількість елементів у масиві вказівників ArrayName.

Щоб встановити значення масиву в деякий метод класу A, можна

використати таку загальну форму

```
ArrayName[index] = &A::SomeMethod;
```

де ArrayName – ім'я масиву вказівників;
index – позиція (індекс) в масиві вказівників;
A – клас, на методи якого вказує масив вказівників;
SomeMethod – деякий метод класу A.

Можна одразу оголосити масив вказівників на методи деякого класу. Загальна форма такого оголошення:

```
type (A:: * ArrayName[size])(arguments_list);
```

де type – тип, що повертають методи класу на які оголошується масив вказівників;
ArrayName – ім'я масиву вказівників;
size – розмір масиву ArrayName;
arguments_list – список аргументів, що отримують методи класу, на які оголошується масив вказівників.

Вказівник на статичний член даних класу

Щоб оголосити вказівник на статичний член даних класу потрібно просто описати вказівник на тип даних, який має статичний член даних. Потім, у деякому програмному коді, цьому вказівнику потрібно присвоїти адресу статичного члена даних об'єкту цього класу.

Послідовність кроків наступна:

- оголосити клас зі статичним членом деякого типу;
- описати статичний член даних класу за межами класу та ініціалізувати його при необхідності;
- у деякому методі або за межами усіх методів (глобальний вказівник) описати вказівник на тип, що має статичний член даних класу;
- описати об'єкт класу зі статичним членом класу;
- присвоїти вказівнику адресу статичного члена даних об'єкту класу.

Приклад. У прикладі оголошується клас CPi зі статичним членом даних Pi. Також оголошується декілька статичних членів, які не є членами класу. У функції _tmain() демонструється використання вказівників на оголошені статичні члени даних.

```
#include «stdafx.h»
#include <iostream>
using namespace std;

// оголошення класу, що містить статичний член даних
class CPi
```

```
{
   public:
   static double Pi; // статичний член даних - константа
};

// оголошення статичної змінної за межами класу з одночасною ініціалізацією
static double Exp = 2.7182818284; // експонента

// ініціалізація статичного члена класу CPi
double CPi::Pi = 3.141592653589793;

// неініціалізовані статичні змінні, значення яких = 0 (за замовчуванням)
static double ZeroD;
static int ZeroI;
static char ZeroC;

int * pZ; // зовнішній вказівник на int

int _tmain(int argc, _TCHAR* argv[])
{
   double d;
   CPi obj; // об'єкт класу
   double * pPi; // вказівник на double
   double * pExp; // вказівник на double

   // вказівник вказує на статичний член даних об'єкту obj класу CPi
   pPi = &obj.Pi;
  pExp = &::Exp; // вказує на статичний член даних Exp - інший вид доступу через ::

   // перевірка
   d = obj.Pi; // d = 3.14159
   d = Exp; // d = 2.71828
   d = ::Exp; // d = 2.71828 - так теж можна
   d = *pPi; // d = 3.14159

   // зовнішній вказівник на int
   pZ = &ZeroI;
   d = *pZ; // d = 0

   *pZ = 5;
   d = ZeroI; // d = 5
```

```
    return 0;
}
```

Якщо статичний член не ініціалізований, то встановлюється значення за замовчуванням:

- для числових типів (int, float, double …) значення неініціалізованого статичного члена рівне 0
- для типу char значення неініціалізованого статичного члена рівне символу з кодом 0.
- для типу bool значення рівне false, що також відповідає значенню 0.

Використання членів даних класу

Дані і методи класу є членами класу. Під час оголошення даних можна застосовувати усі відомі способи доступу. Дані можуть бути оголошені як: приватні (private), захищені (protected), загальнодоступні (public).

Під час оголошення дані ініціалізувати **не можна**. Ініціалізувати дані можна в методах класу, конструкторах класу, зовнішніх методах тощо. Клас – це не об'єкт, і пам'ять під нього не виділяється до тих пір, поки не буде створено екземпляру (об'єкту) класу.

Статичний член даних в класі оголошується з допомогою ключового слова static. Оскільки, це є статичний член даних, то потрібно його визначити (описати змінну) за межами класу. Доступ до цього статичного члена даних мають усі екземпляри (об'єкти) класу.

Щоб використовувати статичний член даних у класі, потрібно:

- оголосити статичний член даних у тілі класу (у будь-якому розділі класу);
- визначити статичний член даних за межами класу.

Приклад. У native-класі (unmanaged-класі) CMyClass оголошується статичний член даних з іменем d цілого типу.

```
// оголошення native-класу
class CMyClass
{
   static int d; // статичний член даних - тільки оголошення

   public:
   CMyClass(void); // конструктор
   ~CMyClass(void); // деструктор

   // методи доступу до статичного члена даних
   int Get(void) { return d; }
   void Set(int nd) { d = nd; }
};
```

```
// визначення статичного члена даних за межами класу
int CMyClass::d; // виділяється пам'ять

// явно заданий конструктор за замовчуванням
CMyClass::CMyClass(void)
{
   d = 0;
}

// деструктор
CMyClass::~CMyClass(void)
{
}
```

Як видно з вищенаведеного коду, з допомогою оператора розширення доступу '**::**' відбувається визначення статичного члена даних.

Використання статичного члена d, оголошеного в класі CMyClass з іншого методу (програмного коду):

```
// використання статичного члена класу
CMyClass MC1; // об'єкт (екземпляр) класу 1
CMyClass MC2; // об'єкт (екземпляр) класу 2

MC2.Set(25);
int t;
t = MC1.Get(); // t = 25 - обидва об'єкти мають доступ до однієї і тієї ділянки пам'яті
```

Якщо в класі оголошено статичний член, то усі екземпляри (об'єкти) класу розділяють цей статичний член класу. Точніше кажучи, цей статичний член класу є спільним для усіх об'єктів цього класу. На відміну від автоматичних даних (даних без слова static), не може створюватись декількох копій статичних даних.

Статичний член даних розподіляється у фіксованій області даних. Це відбувається на стадії компоновки.

Пам'ять для статичного члена даних виділяється на стадії компоновки тільки **один раз** в той момент, коли визначається статичний елемент. У прикладі вище пам'ять виділяється у рядку

```
int CMyClass::d;
```

Можна ініціалізувати статичний член класу при його оголошенні в native-класі. Ініціалізація статичного члена даних класу здійснюється при його оголошенні. Наприклад:

```
int CMyClass::d = 10; // ініціалізація статичного члена даних
```

Якщо статичний член даних оголошений у розділі public, то доступ до нього можна отримати одним з трьох способів:

- з допомогою символу ‘ **.** ‘ (крапка), якщо оголошено об'єкт (екземпляру) класу;
- з допомогою послідовності символів ‘->’, якщо оголошено вказівник на об'єкт класу;
- з допомогою імені класу та символів ‘::’ (розширення області видимості).

Цей спосіб працює також і для статичний членів даних, оголошених в розділах private та protected.

Статичні члени даних класу можуть використовуватись для зв'язування об'єктів класу між собою. Вони є тією спільною ділянкою пам'яті, яку можуть використовувати різні об'єкти класу, що оголошені в різних методах. Пам'ять для статичного члену класу завжди виділена, навіть, якщо в деякому методі не оголошено жодного об'єкту.

Наприклад, можна оголосити клас, що містить статичний член даних, який підраховує кількість активних об'єктів класу. У цьому випадку реалізуються методи, що збільшують (зменшують) значення цього статичного члена. Ці методи можуть викликатись з різних частин програмного коду (методів інших класів тощо). Однак, пам'ять, що виділена для цього статичного члена даних буде спільною для усіх цих методів.

Якщо статичний член класу оголошений в розділі private, то доступ до нього можна отримати одним з трьох способів:

- з допомогою оператора розширення області видимості ‘::’;
- з допомогою функції-члена класу (методу класу);
- з допомогою класу, що оголошений дружнім до даного класу.

Якщо для доступу до прихованого статичного члена даних використати оператор ‘ **.** ‘ або доступ за вказівником ‘**->**‘, то компілятор видасть помилку.

Оголошення та використання об'єкту класу, який є членом-даних іншого класу.

Як відомо, у класах можуть оголошуватись члени даних класу, які є об'єктами інших класів.

Якщо в класі A оголошується об'єкт класу B, то при оголошенні об'єкту класу A конструктори обох класів викликаються у такій послідовності:

- конструктор класу B;
- конструктор класу A.

Таку послідовність дій компілятор формує автоматично. Пам'ять для членів-даних класу, які є об'єктами класу, виділяється при виділенні пам'яті для класу, в якому ці об'єкти реалізовані. Тобто, якщо описати об'єкт класу A, в якому є об'єкт класу B, то пам'ять для об'єкту класу A виділяється з врахуванням

(включає в себе) пам'яті об'єкту класу B. Наприклад. Якщо взяти клас CMyPoint, що оголошений у п. 1 цієї теми, то при створенні об'єкту класу CMyLine, спочатку викликається відповідний конструктор класу CMyPoint (виділяється пам'ять), а потім викликається конструктор класу CMyLine.

Посилання на об'єкт класу

Посилання на клас оголошується з допомогою символу &. При оголошенні змінної, що є посиланням, потрібно одразу її ініціалізувати значенням об'єкту, для якого вже виділена пам'ять.

Загальний вигляд оголошення посилання на об'єкт класу має вигляд:

```
CMyClass & ref = obj;
```

де CMyClass – ім'я класу;

ref – ім'я змінної, яка є посиланням на об'єкт obj;

obj – ім'я об'єкту, для якого виділена пам'ять.

Посилання (змінна-посилання) на об'єкт класу може бути членом даних іншого класу. При оголошенні змінної-посилання в класі, ця змінна має бути ініціалізована одразу в спеціально розробленому конструкторі. У цьому випадку для змінної-посилання класу пам'ять виділяється динамічно.

У прикладі нижче демонструється застосування класу CLine (відрізок), в якому оголошуються два посилання на клас CPoint, що описує точку.

У класі CPoint реалізовано такі члени даних та методи:

- внутрішні змінні x, y – координати точки;
- конструктор за замовчуванням;
- методи доступу GetXY(), SetXY().

У класі CLine оголошуються такі члени даних та методи:

внутрішні змінні-посилання типу CPoint& (посилання на об'єкт класу CPoint) з іменами p1, p2;

конструктор з двома параметрами, що динамічно ініціалізує посилання p1, p2 при оголошенні об'єкту класу;

методи доступу GetPoints(), SetPoints().

Оголошення класів CPoint та CLine має вигляд:

```
// клас, що реалізує точку
class CPoint
{
   int x,y;

   public:
   // конструктор за замовчуванням
   CPoint() { x = y = 0; }

   // методи доступу
```

```
    void GetXY(int* nx, int* ny)
    {
      *nx = x;
      *ny = y;
    }

    void SetXY(int nx, int ny)
    {
      x = nx;
      y = ny;
    }
};

// клас, що реалізує відрізок
class CLine
{
  // посилання на об'єкти типу CPoint
  CPoint & p1;
  CPoint & p2;

  public:

  // конструктор за замовчуванням
  // динамічно ініціалізуються значення посилань p1 та p2
  CLine():p1(* new CPoint), p2(* new CPoint)
  {
    p1.SetXY(0, 0);
    p2.SetXY(1, 1);
  }

  // методи доступу
  // повернути координати точок
  void GetPoints(CPoint* pt1, CPoint* pt2)
  {
    int x, y;
    p1.GetXY(&x, &y); // взяти значення x, y для точки p1
    pt1->SetXY(x, y); // встановити x,y в нову точку pt1

    p2.GetXY(&x, &y); // взяти x,y
    pt2->SetXY(x, y); // записати x,y в pt2
  }

  // встановити нові значення точок
```

```
    void SetPoints(CPoint* pt1, CPoint* pt2)
    {
      int x, y;

      // p1 => pt1
      p1.GetXY(&x, &y);
      pt1->SetXY(x, y);

      // p2 => pt2
      p2.GetXY(&x, &y);
      pt2->SetXY(x, y);
    }
};
```

Важливо: при оголошенні посилання на об'єкт деякого класу, обов'язково потрібно ініціалізовувати це посилання деяким значенням, наприклад, у спеціально розробленому конструкторі за замовчуванням. Якщо не вказати код ініціалізації посилання в конструкторі, то компілятор видасть помилку:

p1' : must be initialized in constructor base/member initializer list ...

У класі CLine здійснюється динамічна ініціалізація посилань p1 та p2 з допомогою конструктора:

```
// динамічна ініціалізація змінних-посилань p1, p2 класу CLine
CLine():p1(* new CPoint), p2(* new CPoint)
{
  // ...
}
```

Використання класу CLine в іншому методі:

```
// посилання на об'єкти як члени-даних класу
CLine cl1; // викликається конструктор, який ініціалізує змінні-посилання p1, p2

// додаткові змінні
CPoint point1, point2;
int x, y;

// перевірка, як конструктор заповнив коодинати точок відрізка
cl1.GetPoints(&point1, &point2);
point1.GetXY(&x, &y); // x = 0; y = 0
point2.GetXY(&x, &y); // x = 1; y = 1
```

```
// встановити нові значення
point1.SetXY(3, 8); // записати точку (3; 8)
point2.SetXY(5, 9); // записати точку (5; 9)

cl1.SetPoints(&point1, &point2);

// перевірка
CPoint pp1, pp2;
cl1.GetPoints(&pp1, &pp2);
pp1.GetXY(&x, &y); // x = 3; y = 8
pp2.GetXY(&x, &y); // x = 5; y = 9
```

Види функцій-членів класу

У класах можна оголошувати такі види функцій:

- "звичайні" **функції-члени**. Це функції, в оголошенні яких не використовуються додаткові ключові слова (const, volatile, static);
- **статичні** функції. Це функції, які оголошені з ключовим словом static;
- **константні** функції. Це функції, які оголошені з ключовим словом const;
- функції-члени, що повертають непостійний об'єкт. Це функції, які оголошені з ключовим словом volatile;
- функції-члени з константним вказівником this;
- функції-члени з непостійним вказівником this;
- вбудовані функції-члени (inline).

Основні відмінності статичних функцій-членів від звичайних:

- викликати статичну функцію-член можна без посилання на об'єкт класу, наприклад CMyClass::MyStaticFuntion(…). У цьому випадку використовується оператор розширення області видимості **'::'**;
- статичні функції-члени не отримують вказівника this класу. Це означає, що вони не можуть отримати доступ до членів даних класу (за винятком членів-даних класу, які оголошені як статичні – з ключовим словомstatic);
- у C++ статична функція-член класу є глобальною. Тому її доцільно використовувати для зміни значень глобальних змінних, які оголошені за межами класу але мають застосування в класі.

Статичні функції-члени класу доцільно застосовувати у випадках, коли в класі потрібно використовувати глобальні змінні, що є спільними для різних об'єктів (екземплярів) цього класу.

Константні функції-члени. Функції, які повертають константний об'єкт називаються константними функціями. Якщо такі функції оголошені в класі, то ці функції називаються константними функціями-членами класу.

Щоб оголосити функцію, яка повертає константний об'єкт, потрібно перед

оголошенням функції розмістити ключове слово 'const'

```
const returned_type FunName(parameters)
{
   // function's body
   // ...
}
```

де returned_type – тип, що повертається функцією;
parameters – параметри функції;
FunName – ім'я функції, що повертає константний об'єкт (константної функції).

Якщо функція оголошена в класі, то приблизне її оголошення буде таким:

```
class CMyClass
{
   private:
   // приватні члени та методи класу
   // ...

   public:
   // загальнодоступні члени та методи класу
   // ...

   // оголошення константної функції в класі
   const returned_type FunName(parameters);
}

// реалізація функції FunName
const returned_type CMyClass::FunName(parameters)
{
   // тіло функції FunName()
   // ...
}
```

Функції-члени volatile. Об'єкт (змінна), що оголошений зі специфікатором volatile може бути змінений у програмі неявно без застосування явно заданих команд. Ключове слово volatile інформує компілятор, що значення змінної у програмі може бути змінене неявно (програмою обробки переривань, фоновим процесом, тощо). Знаючи про неявну мінливість змінної у програмі (ключове слово volatile), компілятор сформує код, що буде опитувати значення змінної перед кожним її використанням у програмі. Таким чином, буде сформовано реальне значення змінної (а не те, що було до зміни).

Наприклад. Нехай у деякій програмі P реалізована глобальна змінна X, що містить адресу, яка може бути змінена програмою-обробником переривання. Програма-обробник переривання викликається незалежно від виконання даної програми P і відповідним чином змінює значення цієї глобальної змінної X. Якщо у програмі P не вказати ключового слова volatile перед X, то, при зчитуванні значення X у програмі P може відображатись старе значення X. Якщо вказати слово volatile, то компілятор буде оновлювати значення змінної X при кожному звертанні до неї, і, спотворення результату не буде.

Функції-члени, які оголошені зі специфікатором volatile просто повертають непостійний (volatile) об'єкт. Такий об'єкт може бути присвоєний змінній, що оголошена з специфікатором volatile.

Загальний вигляд класу, що містить функцію, яка повертає volatile-значення

```
class CMyClass
{
   private:
   // приватні члени та методи класу
   // ...

   public:
   // загальнодоступні члени та методи класу
   // ...

   // оголошення константної функції в класі
   volatile returned_type FunName(parameters);
}

// реалізація функції FunName
volatile returned_type CMyClass::FunName(parameters)
{
   // тіло функції FunName()
   // ...
}
```

де returned_type – тип, що повертається функцією;
parameters – параметри функції;
FunName – ім'я функції, що повертає volatile-об'єкт.

Загальна форма оголошення функції з константним вказівником this

Оголошення функції у класі (на прикладі функцій MyFun1(), MyFun2()) з константним вказівником this має такий загальний вигляд:

```
class CMyClass
```

```
{
  private:
  // приховані члени даних та методи класу
  // ...

  public:
  // функція оголошена з константним вказівником this
  return_type MyFun1(parameters) const; // out-of-line
  return_type MyFun2(parameters) const // inline реалізація
  {
    // тіло функції
    // ...
  }
}

// реалізація out-of-line функції
return_type CMyClass::MyFun1(parameters) const
{
  // тіло функції MyFun1()
  // ...
}
```

де return_type – тип, що повертається тією чи іншою функцією;
parameters – параметри, які отримує функція.

У вищенаведеному фрагменті, у класі CMyClass оголошуються дві функції з константним вказівником this. Функція MyFun1() реалізована за межами класу (типу out-of-line). Функція MyFun2() реалізована в класі (типу inline). Перед тілом кожної функції стоїть ключове слово const.

В чому полягає суть оголошення функції з константним вказівником this? У функції з константним вказівником this перед тілом функції ставиться ключове слово const. Це означає, що тіло функції має обмежені можливості використання.

Слід зауважити, що у цьому випадку ключове слово const не має відношення до значення, що повертається функцією. Воно має значення тільки для вказівника this, що використовується у функції.

Якщо функція оголошена з константним вказівником this, то в тілі функції заборонено змінювати дані класу. При спробі змінити дані класу, буде виникати помилка компіляції.

Відмінність між звичайною функцією та функцією з константним вказівником this добре видно в реалізації нижченаведеного класу.

Нехай задано клас CRadius, що реалізує радіус деякого об'єкта чи геометричної фігури. У класі реалізовано:

- внутрішній член даних типу double;
- конструктор CRadius();

• звичайну функцію GetRadius(), яка збільшує радіус удвічі і повертає його значення;

• функцію GetRadius2(), оголошену з константним вказівником this. Дана функція повертає значення внутрішньої змінної radius.

```
// клас CRadius
class CRadius
{
   double radius;

   public:
   CRadius(void); // конструктор класу за замовчуванням

   // Звичайна функція-член класу,
   // цій функції неявно передається вказівник: CRadius * const this
   double GetRadius(void)
   {
      radius *= 2.0; // дозволено
      return radius;
   }

   // Функція-член класу, оголошена з const
   // цій функції передається вказівник: const CRadius * const this
   double GetRadius2(void) const
   {
      //radius = 2.0; // Заборонено! Помилка компіляції
      return radius;
   }
};
```

Якщо у функції GetRadius2() спробувати змінити значення внутрішньої змінної radius, то вийде помилка компіляції:

Error: 'radius' cannot be modified because it is being accessed through a const object

Ключове слово const перед тілом функції в класі означає, що функції заборонено вносити будь-які зміни в члени даних класу.

Використання класу CRadius в деякому програмному коді, наприклад, обробнику події:

```
CRadius CR;

double d;
d = CR.GetRadius2(); // d = 1.0 - виклик функції з константним this
```

```
d = CR.GetRadius(); // d = 1.0 - виклик звичайної функції-члена
```

Вказівник this у класі є невидимий, функціям класу він передається неявно. Це означає, що при спробі явного опису вказівника this у класі, компілятор видасть помилку. Звичайній функції-члену класу GetRadius() вказівник this передається в неявному вигляді як:

```
CRadius * const this;
```

Функції-члену GetRadius2() невидимий вказівник this передається з ключовим словом const:

```
// модифікований вказівник this
const CRadius * const this;
```

Запис

```
const CRadius
```

означає, що об'єкт типу CRadius є константним об'єктом. І тому, змінювати значення об'єкту у тілі функціїGetRadius2() не можна. Отже, не можна змінювати значення внутрішніх членів-даних класу у функціїGetRadius2(). Спроба змінити значення внутрішнього члену даних radius у тілі функції GetRadius2() викличе помилку компіляції.

Функції з константним вказівником this доцільно використовувати у випадках, коли потрібно, щоб функція-член базового класу випадково не перевизначилась у похідних класах. У цьому випадку константний вказівник thisслужить захистом даних базового класу від випадкової їх зміни у похідних класах.

Якщо ключове слово const розмістити перед іменем функції, а не перед тілом функції, то функція повертає константне значення. Але змінювати внутрішні дані класу у тілі функції можна. Ключове слово const перед іменем функції відноситься до значення, яке повертається функцією. Ключове слово const перед тілом функції відноситься до вказівника this.

Оголошення статичної функції-члена з ключовим словом const не має змісту. При виклику статичної функції-члена, невидимий вказівник this не передається цій функції. При спробі оголошення статичної функції-члена з константним вказівником this компілятор видасть помилку

... modifiers not allowed on static members functions

Загальна форма оголошення функції з непостійним вказівником this

Для того, щоб оголосити функцію-член з непостійним вказівником this використовується ключове слово volatile. Загальна форма оголошення функції з непостійним вказівником this в класі має приблизно наступний вигляд:

```
class CMyClass
{
   // приховані члени даних та методи класу
   // ...

   public:
   // загальнодоступні члени даних та методи класу
   // функція-член з непостійним вказівником this
   return_type MyFun1(parameters) volatile; // out-of-line
   return_type MyFun2(parameters) volatile // inline
   {
      // тіло функції
      // ...
   }
}

// реалізація out-of-line функції
return_type CMyClass::MyFun1 volatile
{
   // тіло функції MyFun1()
   // ...
}
```

де return_type – тип, що повертається тією чи іншою функцією;
parameters – параметри, які отримує функція.

У вищенаведеному фрагменті, у класі CMyClass оголошуються дві функції з непостійним (volatile) вказівником this. Функція MyFun1() реалізована за межами класу (типу out-of-line). Функція MyFun2()реалізована у класі (типу inline). Перед тілом кожної функції стоїть ключове слово volatile.

Оголошення функції як volatile доцільно використовувати у випадках, коли потрібно обробляти об'єкт класу різними процесами, до яких можна віднести:

- обробка об'єкту процесором;
- обробка з допомогою фонових прикладних програм;
- обробка з допомогою програм обробки переривань.

Функція-член класу, що оголошена з ключовим словом volatile трактується компіляторомурізнихвипадкахпорізному.Длятакихфункційкомпіляторвідключає деякі види оптимізації в залежності від типу оголошеного об'єкту. Функції-члени класу, що оголошені з ключовим словом volatile можна використовувати як для автоматичних об'єктів так і для непостійних (volatile) об'єктів. Для константних об'єктів використання функцій з непостійним вказівником заборонено. Знову ж таки, все залежить від налаштувань та реалізації компілятора.

Контрольні завдання та запитання

1. Як оголосити вказівник на об’єкт?
2. Що таке вказівник this?
3. Як звернутися у програмі до елементів масиву об’єктів?
4. Як отримати доступ елементу масива об’єктів до відкритих членів класу?
5. Якими способами можна виконати ініціалізацію масиву об’єктів?
6. Як оголосити двовимірний масив об’єктів?
7. Що таке посилання? Як його оголосити?
8. У чому полягає відмінність параметрів-посилань від звичайних параметрів функцій?

ТЕМА 3. ВБУДОВАНІ ТА ДРУЖНІ ФУНКЦІЇ

Зміст:

inline функції-члени класу
Приклад оголошення класу, що містить вбудовані функції (inline)
Дружні класи та дружні функції
Приклад оголошення класу, що є дружнім до іншого класу
Приклад оголошення функції, що є дружньою до іншого класу

inline функції-члени класу

У мові C++ в класах можуть використовуватись два типи функцій:

- функції, які підставляються або вбудовані функції. Такі функції ще називаються inline-функціями;
- звичайні функції, код яких не підставляється безпосередньо у тіло викликаючої програми.

Inline-функції обробляються так само як макрос. При виклику такої функції з викликаючого коду, тіло функції безпосередньо вставляється у цей код. Інакше кажучи, код inline-функції підставляється у те місце рядка програми, з якого вона викликається. У результаті, виклик inline-функції дає виграш у часі виконання програми (часі обробки функції). Це пов’язано з тим, що зникають накладні витрати на додаткову обробку при передачі (отриманні) параметрів у функцію.

Існує два способи оголошення (створення) inline-функції.

Спосіб 1. З використанням модифікатора inline. Загальний вигляд функції, оголошеної з модифікатором inline наступний:

```
inline returned_type FunName(parameters)
{
   // ...
}
```

де returned_type та parameters відповідно тип та параметри, що повертаються функцією.

Не всі компілятори підтримують цей спосіб.

Спосіб 2. Реалізація коду функції-члена класу безпосередньо в тілі оголошення класу. У цьому випадку використання модифікатора inline необов’язкове.

Загальний вигляд класу з оголошеними inline-функціями має вигляд:

```
class CMyClass
{
   // ...

   // без модифікатора inline
```

```
    returned_type MyInlineFun1(parameters)
    {
       // ...
    }

    // з модифікатором inline
    inline returned_type MyInlineFun2(parameters)
    {
       // ...
    }

    // ...
}
```

де returned_type – тип, що повертається функціями;
parameters параметри, що отримує функція.

Основна перевага використання вбудованих (inline) функцій – це пришвидшення часу виконання програми. Це пов'язано з тим, що при виклику inline-функції не витрачається час на:

- запис аргументів у стек;
- читання аргументів зі стеку при поверненні з функції.

Додавати модифікатор inline в оголошення функції доцільно у випадках, коли виконуються дві основні умови:

- виклик функції відбувається настільки часто, що негативно впливає на швидкість виконання програми або просто уповільнює виконання програми. Наприклад, функція може викликатись багатократно в операторі циклу;
- об'єм коду функції є малим. Функції, що мають великі об'єми програмного коду суттєво збільшують розмір самої програми, що, інколи, теж небажано. Тому в якості вбудованих, рекомендовано використовувати тільки дуже малі функції.

Оголошення inline-функції є запитом а не командою. Тому, компілятор може не виконати запит на генерування коду для оголошеної inline-функції. У цьому випадку, функція може бути використана як звичайна (не inline).

Наприклад:

- у більшості випадків рекурсивні функції не можуть бути використані як inline-функції;
- не можна згенерувати inline функцію, яка містить статичні члени даних.

Приклад оголошення класу, що містить вбудовані функції (inline)

У прикладі оголошується клас CMyPoint. Клас містить внутрішні члени даних а також inline-функцію GetY(). Дві інші функції класу є звичайними, тому що реалізація цих функцій винесена за межі класу.

```
// клас, що містить inline-функції
```

```
class CMyPoint
{
   int x, y;

   public:
   // конструктор класу
   CMyPoint(void);

   int GetX(void); // звичайна (не inline) функція класу
   int GetY(void) // inline-функція
   {
      return y; // реалізація у тілі класу
   }

   // звичайна (не inline) функція
   void SetXY(int nx, int ny);
};

// конструктор класу
CMyPoint::CMyPoint(void)
{
   x = y = 0;
}

// реалізація не inline-функції GetX()
int CMyPoint::GetX(void)
{
   return x;
}

// реалізація не inline функції SetXY()
void CMyPoint::SetXY(int nx, int ny)
{
   x = nx;
   y = ny;
}
```

Використання класу в іншому програмному коді (наприклад, обробнику події)

```
CMyPoint MP1; // об’єкт класу, що містить inline-функцію

// виклик звичайної (не inline) функції
```

```
MP1.SetXY(25,30);

int tx, ty;
tx = MP1.GetX(); // виклик не inline функції
ty = MP1.GetY(); // виклик inline функції
```

Дружні класи та дружні функції

Бувають випадки, коли для заданого класу потрібно оголосити інший клас або функцію, які повинні мати необмежений доступ до внутрішніх змінних та методів класу. Така необхідність виникає з суті задачі, що розв'язується.

Якщо клас А оголошується "дружнім" до класу В, то об'єкти класу А мають доступ до усіх членів даних і методів класу В. Якщо функція оголошується "дружньою" до деякого класу, то у цій функції також є необмежений доступ до членів даних та методів цього класу.

Щоб оголосити "дружній" клас до даного класу, використовується ключове слово friend. Загальна форма оголошення "дружнього" класу до даного має вигляд:

```
class CClass
{
   // ...

   friend class CFriendClass;

   // ...
};

class CFriendClass
{
   // ...
};
```

де CClass – клас, в якому оголошується "дружній" клас CFriendClass. Усі змінні (навіть і private) та методи цього класу є доступними для об'єктів класу CFriendClass;

CFriendClass – клас, який є "дружнім" до класу CClass. Оголошення "дружнього" класу CFriendClass до класу CClass може бути в будь-якому місці тіла класу – у межах оголошення класу (між фігурними дужками { }).

Оголошення "дружньої" функції до класу починається з ключового слова friend. Загальна форма оголошення "дружньої" функції до класу має вигляд:

```
friend type FunName(parameters);
```

де FunName – ім'я "дружньої" функції;

type – тип, що повертається функцією FunName();

parameters – параметри "дружньої" функції. Щоб отримати об'єкт потрібного класу у функції FunName()доцільно передати у цю функцію посилання (або покажчик) на об'єкт цього класу.

Якщо потрібно оголосити "дружню" функцію у деякому класі, то загальний вигляд такого оголошення наступний:

```
class CClass
{
   // ...

   friend type FunName(parameters);

   // ...
};
```

де FunName – ім'я "дружньої" функції;

type – тип, що повертається функцією FunName();

parameters – параметри "дружньої" функції.

Оголошувати "дружній" клас або функцію до заданого класу можна у будь-якому місці чи розділі класу в межах його оголошення (між фігурними дужками { }).

"Дружніх" функцій та "дружніх" класів можна оголошувати у тілі класу скільки завгодно.

Щоб отримати об'єкт потрібного класу у функції доцільно передати у цю функцію посилання (або покажчик) на об'єкт цього класу.

Наприклад.

Нехай задано клас з іменем CMyClass. Потрібно оголосити "дружню" функцію до цього класу, яка має ім'яFriendFun(). Функція повертає параметр типу int.

Оголошення класу CMyClass та "дружньої" функції у класі має вигляд:

```
// оголошення класу CMyClass, в якому є «дружня» функція FriendFun()
class CMyClass
{
   // тіло класу
   // ...

   friend int FriendFun(CMyClass &);

   // ...
};
```

Реалізація “дружньої” функції FriendFun():

```
int FriendFun(CMyClass & mc)
{
   // використання об’єкту класу mc для доступу до членів класу CMyClass
   // ...
};
```

Приклад оголошення класу, що є дружнім до іншого класу

Нехай задано клас Number, що містить цілочисельну величину. Також задано клас RangeNum, який містить величину Number але в межах заданого діапазону.

Щоб з класу RangeNum можна було мати доступ до приватної змінну num класу Number, клас RangeNumоголошується “дружнім” до класу Number.

Лістинг класів Number та RangeNum має вигляд:

```
// клас, що реалізує ціле число
class Number
{
   // оголошення дружнього класу RangeNum до класу Number
   friend class RangeNum;
   int num;

   public:
   // конструктори
   Number() { num = 0; }
   Number(int num) { this->num = num; }
};

// оголошення класу RangeNum, який тримає число Number в заданих межах
class RangeNum
{
   Number num; // об’єкт класу Number - просте ціле число
   int min; // нижня межа числа num
   int max; // верхня межа числа num

   public:
   // конструктор класу
   RangeNum()
   {
      // доступ до private-члена класу Number, тому що RangeNum є дружнім до Number
       num.num = 0;
```

```
            // встановлення діапазону 0..99 за замовчуванням
            min = 0;
            max = 99;
        }

        // методи доступу
        int GetNum(void)
        {
             return num.num; // доступ до приватного члена з «дружнього» класу
Range
        }

        void SetNum(int nnum)
        {
            num.num = nnum; // доступ до приватного члена з «дружнього» класу
Range
            if (num.num>max) num.num = max-1;
            if (num.num<min) num.num = min;
        }

        // встановлення діапазону для num в заданих межах
        void SetRange(int min, int max)
        {
            this->min = min;
            this->max = max;
           if (num.num>max) num.num = max-1; // знову доступ через дружній клас
            if (num.num<min) num.num = min;
        }
    };
```

Використання класів у іншому програмному коді

```
    int _tmain(int argc, _TCHAR* argv[])
    {
        // об’єкт класу Range
        RangeNum r;
        int d;
        d = r.GetNum(); // d = 0

        r.SetRange(100, 200);
        r.SetNum(101);
        d = r.GetNum(); // d = 101
```

```
    r.SetRange(10, 20); // корегується значення num
    d = r.GetNum(); // d = 19
    r.SetNum(-10);
    d = r.GetNum(); // d = 10

    return 0;
}
```

Якщо у класі Number в оголошенні “дружнього” класу RangeNum

```
friend class RangeNum;
```

забрати ключове слово friend, то у конструкторі та усіх методах класу при доступі до num.num компілятор видасть помилку:

Number::num: cannot access private member declared in class ‘Number’

Приклад оголошення функції, що є дружньою до іншого класу

У прикладі оголошується клас Radius, що містить величину радіуса деякої геометричної фігури. У класі оголошуються:

- одна прихована (private) змінна radius;
- методи Get() та Set() для доступу до змінної radius;
- дві зовнішні “дружні” функції GetLength() та GetArea();
- один “дружній” клас Volume. У класі Volume оголошується зовнішній (public) метод GetVolume(), який повертає об’єм кулі заданого радіуса.

Лістинг додатку типу “Win32 Console Application” наступний:

```
#include «stdafx.h»
#include <iostream>

using namespace std;

// клас, що реалізує величину радіуса геометричної фігури
class Radius
{
   private:
   double radius; // прихована змінна radius

   // оголошення «дружніх» функцій - в будь-якому розділі класу Radius
   friend double GetLength(Radius &);
   friend double GetArea(Radius &);

   // оголошення «дружнього» класу
   friend class Volume;
```

```
    public:
    // методи доступу до radius
    double Get(void) { return radius; }
    void Set(double nradius) { radius = nradius; }
};

// «дружні» функції до класу Radius
// довжина кола
double GetLength(Radius & r)
{
    // доступ до private-члена radius класу з «дружньої» функції GetLength()
    return (double)(2 * r.radius * 3.1415);
}

// площа круга
double GetArea(Radius & r)
{
    // доступ до private-члена radius класу з «дружньої» функції GetArea()
    return (double)(r.radius * r.radius * 3.1415);
}

// оголошення «дружнього» класу
class Volume
{
    public:
    // клас містить тільки одну функцію Volume
    double GetVolume(Radius * r) // функція отримує покажчик на Radius
    {
        // доступ до private-члена класу Radius з «дружнього» класу Volume
        return (double)(4.0 / 3.0 * 3.1415 * r->radius * r->radius * r->radius);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    // об’єкт класу Radius
    Radius r;
    Volume v;
    double len, area, vol;

    r.Set(3);

    // виклик зовнішньої «дружньої» функції GetLength()
```

```
    len = GetLength(r); // передача об’єкту класу Radius за посиланням

    // виклик зовнішньої «дружньої» функції GetArea()
    area = ::GetArea(r); // передача за посиланням

    // виклик функції «дружнього» класу v
    vol = v.GetVolume(&r); // передача за покажчиком

    cout << «Length = « << len << endl; // Length = 9.4245
    cout << «Area = « << area << endl; // Area = 28.2735
    cout << «Volume = « << vol << endl; // Volume = 113.094

    return 0;
}
```

Як видно з вищенаведеного коду, “дружні” функції отримують вхідним параметром посилання або покажчик на об’єкт класу, до членів даних та методів якого вони мають необмежений доступ.

Контрольні завдання та запитання

1. Що таке вбудована функція? У чому полягають її переваги та недоліки?
2. Є два способи зробити функцію вбудованою. Що це за способи?
3. Наведіть щонайменше два обмеження на використання вбудованих функцій.
4. Змініть наступну програму так, щоб усі функції-члени за замовчуванням стали вбудованими:

```
#include <iostream.h> class myclass { int i,j; public:
myclass(int i, int j); void show();
}
myclass::myclass(int x, int y) { i=x;
j=y;
}
void myclass::show ( ) {
cout <<i<<"    " << j << "\n";
}
void main( ) { myclass count(2,3); count.show( );
}
```

5. Що таке дружня функція?
6. Чи є дружня функція членом класу?
7. Який механізм роботи дружньої функції із закритими членами класу?
8. Чи може функція бути дружньою до декількох класів?
9. Чи може функція бути дружньою до одного класу і бути членом іншого класу?
10. Чи успадковується дружня функція?

ТЕМА 4. ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ

Зміст:

Визначення терміну “перевантаження” функції
Приклади перевантаження функцій
Приклад перевантаження функції у класі
Умови перевантаження функції
Переваги перевантаження конструкторів класу
Доступ до перевантаженої функції з допомогою покажчика на функцію
Використання ключового слова overload

Визначення терміну “перевантаження” функції

“Перевантаження” функції – це оголошення функції з тим же іменем декілька разів. Таким чином, в деякій області видимості ім’я “перевантажена” функція оголошується декілька разів. Щоб компілятор міг відрізняти “перевантажені” функції, ці функції повинні відрізнятися між собою списком вхідних параметрів.

В загальному випадку оголошення перевантаженої функції в деякій області видимості виглядає наступним чином:

```
return_type1 FunName(parameters_list_1)
{
   // ...
}

return_type2 FunName(parameters_list_2)
{
   // ...
}

return_typeN FunName(parameters_list_N)
{
   // ...
}
```

де FunName – ім’я перевантаженої функції;

parameters_list1, parameters_list2, …, parameters_listN – списки параметрів “перевантаженої” функції з іменем FunName;

return_type1, return_type2, …, return_typeN – типи параметрів, що повертаються “перевантаженою” функцією FunName. Компілятор розрізняє “перевантажені” функції тільки за списком отриманих параметрів але не за повернутим значенням.

Перевантажені функції відрізняються за списком параметрів. Списки параметрів перевантажених функцій мають відрізнятися за такими ознаками:

- кількістю параметрів;
- якщо кількість параметрів однакова, то типами параметрів.

Наприклад. Функція Max() перевантажена і відрізняється кількістю параметрів та типами параметрів.

```
// функція Max з двома параметрами типу int
int Max(int a, int b)
{
   // ...
}

// функція Max з трьома параметрами типу int
int Max(int a, int b, int c)
{
   // ...
}

// функція Max з двома параметрами типу double
int Max(double a, double b)
{
   // ...
}
```

Приклади перевантаження функцій

Приклад 1. Функція Equal(), яка порівнює два значення різних типів. Функція має три перевантажені реалізації для різних типів (char, int, double).

Програмний код, що демонструє використання функції Equal() наступний:

```
#include «stdafx.h»
#include <iostream>
using namespace std;

bool Equal(char c1, char c2)
{
   return (bool)(c1==c2);
}

bool Equal(double d1, double d2)
{
   return (bool)(d1==d2);
}

bool Equal(int i1, int i2)
```

```
{
   return (bool)(i1==i2);
}

int _tmain(int argc, _TCHAR* argv[])
{
   bool b;
   b = Equal(5,6); // викликається Equal(int, int), b = 0
   b = Equal(3.755, 3.755); // викликається Equal(double, double), b = 1
   b = Equal('A', 'A'); // викликається Equal(char, char), b = 1
   return 0;
}
```

Приклад 2. Функція Average(), яка знаходить середнє арифметичне. Функція має 4 перевантажені реалізації для різної кількості цілочисельних параметрів.

Програмний код, що демонструє застосування функції наступний:

```
#include «stdafx.h»
#include <iostream>
using namespace std;

double Average(int a, int b)
{
   return (a+b)/2.0;
}

double Average(int a, int b, int c)
{
   return (a+b+c)/3.0;
}

double Average(int a, int b, int c, int d)
{
   return (a+b+c+d)/4.0;
}

double Average(int a, int b, int c, int d, int e)
{
   return (a+b+c+d+e)/5.0;
}

int _tmain(int argc, _TCHAR* argv[])
{
```

```
    double avg;
    avg = Average(2,3); // avg = 2.5
    avg = Average(2,3,5); // avg = 3.3333
    avg = Average(2,3,5,8); // avg = 4.5
    avg = Average(2,3,5,8,11); // avg = 5.8
    return 0;
}
```

Приклад перевантаження функції у класі

Задано клас Day, що реалізує день тижня. У класі перевантажено функцію Set(), що встановлює новий день тижня. Метод Set() має 2 реалізації:

- без параметрів;
- з одним параметром.

Програмний код, що демонструє застосування “перевантаження” функцій у класі має наступний вигляд:

```
#include «stdafx.h»
#include <iostream>
using namespace std;

class Day
{
   int day;

   public:
   // метод Set() перевантажено
   void Set(int nday) { day = nday; }
   void Set(void) { day = 1; }
   int Get(void) { return day; }
};

int _tmain(int argc, _TCHAR* argv[])
{
   Day D;
   int day;

   // використання перевантаженого метода Set
   D.Set();
   day = D.Get(); // day = 1

   D.Set(5);
   day = D.Get(); // day = 5
```

```
    cout << day << endl;
    return 0;
}
```

Умови перевантаження функції

Компілятор розпізнає перевантажені функції тільки за отримуваними параметрами. Якщо дві функції мають однакові імена, однакову кількість та типи параметрів але повертають різні значення, то такі функції вважаються однаковими. У цьому випадку компілятор видасть помилку:

Наприклад. У нижченаведеному коді оголошується дві функції з іменем Inc5():

```
int Inc5(int d)
{
   return d+5;
}

double Inc5(int d)
{
   return (double)(d+5.0);
}
```

Як видно, функції повертають значення різних типів. Оскільки, функції мають однакові імена, однакову кількість та типи параметрів, то вищенаведений код видасть помилку компілятора

‘double Get5(int)’ : overloaded function differs only by return type from ‘int Get5(int)’

Переваги перевантаження конструкторів класу

Перевантаження конструкторів класу дає такі переваги:

- підвищення гнучкості класу при його створенні. З допомогою перевантаження конструкторів користувач вибирає оптимальний спосіб створення об’єкту. Якщо спробувати створити об’єкт класу способом, для якого непередбачений конструктор, то компілятор видасть повідомлення про помилку;
- можливість створення ініціалізованих та неініціалізованих об’єктів або об’єктів, що ініціалізовані за замовчуванням. У цьому випадку передбачається 2 варіанти реалізації конструктора: з ініціалізацією та без неї;
- можливість створення конструкторів копіювання.

Приклад перевантаження конструкторів у класі

Задано клас Cylinder що реалізує циліндр. У класі перевантажується конструктор, який може викликатись одним з трьох способів:

- конструктор без параметрів;
- конструктор з 1 параметром;

- конструктор з 2 параметрами.

```
#include «stdafx.h»
#include <iostream>
using namespace std;

class Cylinder
{
   double r, h;

   public:
   // конструктор без параметрів
   Cylinder()
   {
      r = 1;
      h = 1;
   }

   // конструктор з 1 параметром
   Cylinder(double h)
   {
      r = 1.0;
      this->h = h;
   }

   // конструктор з двома параметрами
   Cylinder(double h, double r)
   {
      this->h = h;
      this->r = r;
   }

   // методи доступу
   double GetR(void) { return r; }
   double GetH(void) { return h; }
};

int _tmain(int argc, _TCHAR* argv[])
{
   Cylinder cl; // виклик конструктора без параметрів
   Cylinder cl1(5); // виклик конструктора з 1 параметром
   Cylinder cl2(7, 9); // виклик конструктора з 2 параметрами
   double d;
```

```
    d = cl.GetH(); // d = 1
    d = cl1.GetH(); // d = 5
    d = cl2.GetH(); // d = 7

    cout << d << endl;
    return 0;
}
```

Доступ до перевантаженої функції з допомогою покажчика на функцію

При оголошенні покажчика на "перевантажену" функцію, компілятор визначає потрібну функцію для покажчика за його сигнатурою при оголошенні.

Наприклад. Нехай задано 3 "перевантажені" функції Increment() для типів int, double, char. При оголошенні покажчика на функцію, потрібно явно задати тип покажчика при оголошенні.

Нижченаведений код демонструє використання покажчика на перевантажену функцію

```
#include «stdafx.h»
#include <iostream>
using namespace std;

int Increment(int i)
{
   return i+1;
}

double Increment(double d)
{
   return d+1;
}

char Increment(char c)
{
   return c+1;
}

int _tmain(int argc, _TCHAR* argv[])
{
     int (*pi)(int); // покажчик на функцію, яка отримує параметром int i повертає int
    pi = Increment; // p вказує на int Increment(int)
```

```
    int d = (*pi)(4); // d = 5

    char (*pc)(char);
    pc = Increment; // p вказує на char Increment(char)
    char c = (*pc)('F'); // c = 'G'

    cout << c << endl;
    return 0;
}
```

Використання ключового слова overload

Ключове слово overload використовувалось у ранніх версіях C++ для вказання того, що функція є перевантаженою. Загальна форма оголошення "перевантаженої" функції з використання ключового слова overload має вигляд:

```
overload ім'я_функції;
```

Наприклад. Якщо перевантажена функція Max() що знаходить максимум між множиною чисел, то рядок

```
overload Max;
```

сповіщає компілятор про те, що функція Max() є перевантаженою.

Контрольні завдання та запитання

1. Що таке поліморфізм?
2. Які функції називають перевантаженими?
3. Для чого введено перевантаження функцій?
4. Наведіть переваги використання перевантаження конструктора класу.
5. Покажіть, як перевантажити конструктор для наступного класу так, щоб можна було створити неініціалізовані об’єкти. Під час створення неініціалізованих об’єктів потрібно присвоїти x та y значення 0:

```
class myclass { int x,y;
public:
myclass(int i, int j) { x = i;
y = j;
}
... // інші функції класу
};
```

6. Поясніть, що таке аргумент за замовчуванням.
7. Що неправильно в прототипах, які використовують аргументи, що передаються за замовчуванням:

```
int f(int count, int max = count); void func(int x = 4, int y)
```

ТЕМА 5. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

Зміст:

Унарні та бінарні оператори

Суть перевантаження операторів

Приклад перевантаження унарних та бінарних операторів для класу, який містить одиночні дані

Приклад перевантаження оператора '*', що обробляє клас, який містить масив дійсних чисел

Обмеження, які накладаються на перевантажені оператори

Перевантаження операторів +, –, *, / з допомогою "дружніх" операторних функцій

Приклад перевантаження бінарного оператора '–' у класі для якого реалізована "дружня" операторна функція

Унарні та бінарні оператори

Розрізняють три основні види операторів: унарні, бінарні та n-арні. Унарні оператори – це оператори, які для обчислення вимаюгають одного операнда, який може розміщуватись справа або зліва від самого оператора. Приклади унарних операторів: i++, --a, -8.

Бінарні оператори – це оператори, що для обчислення вимагають двох операндів. Наприклад, нижче наведено фрагменти виразів з бінарними операторами +, –, %, *:

```
a+b
f1-f2
c%d
x1*x2
```

n-арні оператори для обчислень потребують більше двох операндів. У мові C++ є тернарна операція ?:, яка для своєї роботи потребує трьох операндів.

Суть перевантаження операторів

Мова C++ має широкі можливості для перевантаження більшості операторів. Перевантаження оператору означає використання оператору для оперування об'єктами класів. Перевантаження оператору – спосіб оголошення та реалізації оператору таким чином, що він обробляє об'єкти конкретних класів або виконує деякі інші дії. При перевантаженні оператору в класі викликається відповідна операторна функція (operator function), яка виконує дії, що стосуються даного класу. Якщо оператор "перевантажено", то його можна використовувати в інших методах у звичному для нього вигляді. Наприклад, команди поелементного сумування двох масивів a1 та a2

```
a1.add(a2);
a3 = add(a1, a2);
```

краще викликати більш природнім способом:

```
a1 = a1 + a2;
```

a3 = a1 + a2;

У даному прикладі оператор ‘+’ вважається перевантаженим.

Для заданого класу операторну функцію в класі можна реалізувати:

- всередині класу. У цьому випадку, операторна функція є методом класу;
- за межами класу. У цьому випадку операторна функція оголошується за межами класу як “дружня” (з ключовим словом friend).

Загальна форма операторної функції, яка реалізована в класі, має такий вигляд:

```
return_type ClassName::operator#(arguments_list)
{
   // деякі операції
   // ...
}
```

де return_type – тип значення, що повертається операторною функцією;

ClassName – ім’я класу, в якому реалізована операторна функція;

operator# – ключове слово, що визначає операторну функцію в класі. Символ # замінюється оператором мови С++, який перевантажується. Наприклад, якщо перевантажується оператор +, то потрібно вказати operator+;

argument_list – список параметрів, які отримує операторна функція. Якщо перевантажується бінарний оператор, то argument_list містить один аргумент. Якщо перевантажується унарний оператор, то список аргументів пустий.

Приклад перевантаження унарних та бінарних операторів для класу, який містить одиночні дані

Операторна функція реалізована всередині класу.

Оголошується клас Point, що реалізує точку на координатній площині. У класі реалізовано:

- дві внутрішні змінні x, y, що є координатами точки;
- два конструктори класу;
- методи доступу до внутрішніх змінних класу GetX(), GetY(), SetX(), SetY();
- дві операторні функції operator+() та operator-(). Операторна функція operator+() перевантажує бінарний оператор ‘+’. Операторна функція operator-() перевантажує унарний оператор ‘-‘.

```
// Клас, що реалізує точку на координатній площині
// клас містить дві операторні функції
class Point
{
private:
   int x, y; // координати точки
```

```
    public:
        // конструктори класу
        Point()
        {
            x = y = 0;
        }

        Point(int nx, int ny)
        {
            x = nx;
            y = ny;
        }

        // методи доступу до членів класу
        int GetX(void) { return x; }
        int GetY(void) { return y; }
        void SetX(int nx) { x = nx; }
        void SetY(int ny) { y = ny; }

        // перевантажений бінарний оператор '+'
        Point operator+(Point pt)
        {
            // p - тимчасовий об'єкт, який створюється з допомогою конструктора без параметрів
            Point p;
            p.x = x + pt.x;
            p.y = y + pt.y;
            return p;
        }

        // перевантажений унарний оператор '-'
        Point operator-(void)
        {
            Point p;
            p.x = -x;
            p.y = -y;
            return p;
        }
    };
```

Як видно з вищенаведеного коду, операторна функція operator+() отримує один параметр. Це означає, що ця функція реалізує бінарний оператор '+'. Цей параметр відповідає операнду, що розміщується у правій частині бінарного

оператора ‘+’. Операнд, що розміщується в лівій частині оператора ‘+’ передається операторній функції неявно з допомогою покажчика this цього класу. Виклик операторної функції здійснює об’єкт, який розміщується в лівій частині оператора присвоювання. Демонстрація використання перевантажених операторів класу Point в іншому методі:

```
// оголошення змінних - об’єктів класу CPoint
Point P1(3,4);
Point P2(5,7);
Point P3;
int x, y; // додаткові змінні

// 1. Використання перевантаженого бінарного оператора ‘+’
P3 = P1 + P2; // об’єкт P1 викликає операторну функцію

// перевірка
x = P3.GetX(); // x = 8
y = P3.GetY(); // y = 11

// 2. Використання перевантаженого унарного оператора ‘-’
P3 = -P2;
x = P3.GetX(); // x = -5
y = P3.GetY(); // y = -7
```

У вищенаведеному коді в операції додавання ‘+’ об’єкт P1 викликає операторну функцію. Тобто фрагмент рядка

```
P1 + P2
```

замінюється викликом

```
P1.operator+(P2)
```

Реалізувати операторну функцію operator+() в класі можна й по іншому

```
Point operator+(Point pt)
{
   // виклик конструктора з двома параметрами
   return Point(x+pt.x, y+pt.y); // створюється тимчасовий об’єкт, який потім копіюється
}
```

У вищенаведеній функції в операторі return створюється тимчасовий об’єкт

шляхом виклику конструктора з двома параметрами, який реалізований в класі. Якщо (у даному випадку) з тіла класу забрати конструктор з двома параметрами

```
// конструктор з двома параметрами
Point(int nx, int ny)
{
   // ...
}
```

то вищенаведений варіант функції operator+() працювати не буде, тому що для створення об'єкту типу Point ця функція використовує конструктор з двома параметрами. У цьому випадку компілятор видасть повідомлення

Point::Point : no overloaded function takes 2 arguments

що означає: немає методу (конструктора) Point::Point() який приймає 2 аргументи.

Приклад перевантаження оператора '*', що обробляє клас, який містить масив дійсних чисел

Операторна функція реалізована всередині класу.

У прикладі реалізується операторна функція operator*(), яка множить поелементно значення внутрішніх масивів об'єктів класу ArrayFloat. Якщо розмір масивів неоднаковий, то перемножується тільки та кількість елементів, яка є мінімальною між двома розмірами масивів.

```
// масив дійсних чисел
class ArrayFloat
{
private:
   float A[10]; // масив дійсних чисел, фіксований розмір масиву
   int size;

public:
   // конструктори
   ArrayFloat()
   {
      size = 0;
   }

   ArrayFloat(int nsize, float nA[])
   {
      size = nsize;
      for (int i=0; i<nsize; i++)
         A[i] = nA[i];
   }
```

```
    // методи доступу
    float GetAi(int i)
    {
       if ((i>=0) && (i<=size-1))
          return A[i];
       else
          return 0;
    }

    void SetAi(int i, float value)
    {
       if ((i>=0) && (i<=size-1))
          A[i] = value;
    }

    // перевантажений оператор '*'
    ArrayFloat operator*(ArrayFloat AF)
    {
       ArrayFloat tmp;
       int n;

       if (size<AF.size)
          n = AF.size;
       else
          n = size;

       for (int i=0; i<n; i++)
          tmp.A[i] = A[i] * AF.A[i];
       tmp.size = n;
       return tmp;
    }
};
```

Використання класу ArrayFloat в іншому методі

```
// додаткові змінні та масиви
float x, y;
float AF1[] = { 2, 5, 7, 9, 12 };
float AF2[] = { 3, 4, 9, 8, 10, 13 };

// створити об'єкти класу ArrayFloat
ArrayFloat A1(5, AF1);
ArrayFloat A2(6, AF2);
```

```
ArrayFloat A3;

// виклик операторної функції operator*
A3 = A1 * A2; // здійснюється поелементне множення

// перевірка
x = A3.GetAi(0); // x = 6
y = A3.GetAi(1); // y = 20
x = A3.GetAi(2); // x = 63
y = A3.GetAi(4); // y = 120
```

Обмеження, які накладаються на перевантажені оператори

На використання перевантажених операторів накладаються наступні обмеження:

- при перевантаженні оператору не можна змінити пріоритет цього оператора;
- не можна змінити кількість операндів оператора. Однак, у коді операторної функції можна один з параметрів (операндів) не використовувати;
- не можна перевантажувати оператори :: . * ?:;
- не можна викликати операторну функцію з аргументами за замовчуванням. Виняток – операторна функція виклику функції operator()().

Не можна перевантажувати наступні оператори:

:: – розширення області видимості;

. (крапка) – вибір члена класу або структури;

* – доступ за покажчиком;

?: – тернарний оператор ?:.

Операторна функція може повертати об’єкти будь-яких типів. Найчастіше операторна функція повертає об’єкт типу класу, в якому вона реалізована або з якими вона працює.

Приклад. Задано клас Complex, в якому перевантажується два оператори:

- унарний оператор ‘+’, який повертає модуль комплексного числа (тип double);
- бінарний оператор ‘+’, який повертає суму комплексних чисел. Операторна функція повертає об’єкт типуComplex;
- бінарний оператор ‘+’, який додає до комплексного числа деяке дійсне число. У цьому випадку операторна функція отримує вхідним параметром дійсне число і повертає об’єкт типу Complex.

Текст класу наступний:

```
// клас Complex
class Complex
{
private:
```

```
    float real; // дійсна частина
    float imag; // уявна частина

  public:
    // конструктори
    Complex(void)
    {
      real = imag = 0;
    }

    Complex(float _real, float _imag)
    {
      real = _real;
      imag = _imag;
    }

    // методи доступу
    float GetR(void) { return real; }
    float GetI(void) { return imag; }

    void SetRI(float _real, float _imag)
    {
      real = _real;
      imag = _imag;
    }

    // оголошення операторної функції, яка перевантажує бінарний '+'
    // функція повертає об'єкт, що містить суму двох комплексних чисел
    Complex operator+(Complex c)
    {
      Complex c2; // тимчасовий об'єкт

      // додавання комплексних чисел
      c2.real = real + c.real;
      c2.imag = imag + c.imag;

      return c2;
    }

    // оголошення операторної функції, яка перевантажує унарний '+'
    // функція повертає модуль комплексного числа
    float operator+(void)
    {
```

```
            float res;
            res = std::sqrt(real*real+imag*imag);
            return res;
        }

        // оголошення операторної функції operator+()
          // функція додає до комплексного числа деяке число, яке є вхідним параметром
        Complex operator+(float real)
        {
            Complex c2; // результуючий об'єкт
            c2.real = this->real + real;
            c2.imag = this->imag;
            return c2;
        }
    };
```

Нижче демонструється використання класу Complex та перевантажених операторних функцій в деякому іншому методі

```
    Complex c1(1,5);
    Complex c2(3,-8);
    Complex c3; // результуючий об'єкт
    double d;

    // перевірка
    c3 = c1 + c2;
    d = c3.GetR(); // d = 1 + 3 = 4
    d = c3.GetI(); // d = 5 + (-8) = -3

    // перевантажений унарний оператор '+'
    d = +c1; // d = |1 + 5j| = 5.09902 - модуль числа
    d = +c2; // d = |3 + (-8)j| = 8.544

    // виклик перевантаженого бінарного '+',
    // додати до комплексного числа число
    c3 = c1 + 5.0;
    d = c3.GetR(); // d = 1 + 5 = 6
```

Можна змінювати значення операндів в операторній функції. Однак такі дії не є корисними з точки зору здорового глузду. Так, наприклад, операція множення 6 * 9 не змінює значення своїх операндів 6 та 9. Результат рівний 54. Якщо операторна функція operator*() буде змінювати значення своїх операндів,

то це може призвести до невидимих помилок в програмах, оскільки програміст за звичкою, буде вважати, що значення операндів є незмінні.

Не можна реалізувати операторні функції в класі, які перевантажують однаковий оператор, отримують однакові параметри але повертають різні значення. Операторна функція не може мати декілька реалізацій в класі з однаковою сигнатурою параметрів (коли типи та кількість параметрів співпадають). У випадку порушення цього правила компілятор видає помилку:

cannot overload functions distinguished by return type alone

що означає

не можна перевантажувати функції, що відрізняються тільки типом повернення

Наприклад. Не можна в класі перевантажувати оператор '+' так як показано нижче

```
class SomeClass
{
   // ...

   SomeClass operator+(SomeClass c1)
   {
      // ...
   }

   double operator+(SomeClass c1) // це є помилка!
   {
      // ...
   }

   // ...
}
```

Це правило стосується будь-яких функцій класу.

Можна реалізувати дві і більше операторних функцій в класі, які перевантажують однаковий оператор і отримують різні (відмінні між собою) параметри.

Перевантаження операторів +, –, *, / з допомогою "дружніх" операторних функцій

Існує два способи перевантаження будь-якого оператору:

з допомогою операторної функції, яка реалізована всередині класу;

з допомогою операторної функції, яка реалізована як "дружня" (friend) до класу.

Між способами реалізації операторних функцій існує відмінність у кількості параметрів, які отримує операторна функція:

- для унарних операторів операторна функція всередині класу не отримує параметрів. А “дружня” до класу операторна функція отримує один параметр;
- для бінарних операторів операторна функція всередині класу отримує один параметр. А “дружня” функція отримує два параметри. У цьому випадку першим параметром “дружньої” функції є лівий операнд, а другим параметром є правий операнд.

Ці відмінності виникають через те, що “дружня” операторна функція не отримує неявного покажчика this. Тому в ній потрібно явно задавати параметри.

Операторна функція може бути реалізована за межами класу. Якщо операторна функція перевантажує унарний оператор, то вона містить один параметр. Якщо операторна функція перевантажує бінарний оператор, то вона містить два параметри. Загальна форма операторної функції, що є дружньою до класу має вигляд

```
return_type operator#(arguments_list)
{
   // деякі операції
   // ...
}
```

де return_type – тип значення, що повертає операторна функція;

operator# – ключове слово, що визначає операторну функцію в класі. Символ # визначає оператор мови C++, який перевантажується. Наприклад, якщо перевантажується оператор +, то потрібно вказати operator+;

argument_list – список параметрів, які отримує операторна функція. Якщо у “дружній” функції перевантажується бінарний оператор, то argument_list містить два аргументи. Якщо перевантажується унарний оператор, то argument_list містить один аргумент.

У класі ця функція має бути оголошена як “дружня” з ключовим словом friend.

Приклад перевантаження бінарного оператора ‘–’ у класі для якого реалізована “дружня” операторна функція

За даним прикладом можна розробляти власні операторні функції, які є “дружніми” до заданого класу. Задано клас Complex, який реалізує комплексне число. У класі оголошуються внутрішні змінні, конструктори, методи доступу та “дружня” функція operator-(). “Дружня” функція operator-(), що реалізована за межами класу, здійснює віднімання комплексних чисел.

```
// клас Complex
class Complex
{
private:
```

```
    float real; // дійсна частина
    float imag; // уявна частина

public:
    // конструктори
    Complex(void)
    {
        real = imag = 0;
    }

    Complex(float _real, float _imag)
    {
        real = _real;
        imag = _imag;
    }

    // методи доступу
    float GetR(void) { return real; }
    float GetI(void) { return imag; }

    void SetRI(float _real, float _imag)
    {
        real = _real;
        imag = _imag;
    }

    // оголошення «дружньої» до класу Complex операторної функції
    friend Complex operator-(Complex c1, Complex c2);
};

// «дружня» до класу Complex операторна функція,
// реалізована за межами класу,
// здійснює віднімання комплексних чисел
Complex operator-(Complex c1, Complex c2)
{
    Complex c; // створити об'єкт класу Complex

    // віднімання комплексних чисел
    c.real = c1.real - c2.real;
    c.imag = c1.imag - c2.imag;

    return c;
}
```

Використання класу Complex в іншому методі

```
// використання «дружньої» операторної функції
Complex c1(5,6);
Complex c2(3,-2);
Complex c3; // результат
float a, b;

// перевірка
a = c1.GetR(); // a = 5
b = c1.GetI(); // b = 6

// виклик «дружньої» до класу Complex операторної функції
c3 = c1 - c2;

// результат
a = c3.GetR(); // a = 5-3 = 2
b = c3.GetI(); // b = 6-(-2) = 8
```

Як видно з вищенаведеного прикладу, “дружня” функція operator-() отримує два параметри. Перший параметр відповідає лівому операнду в бінарному операторі віднімання ‘–’. Другий параметр відповідає правому операнду.

Контрольні завдання та запитання

1. Для чого введено перевантаження операторів?
2. Запишіть загальні форми подання оператора-функції члена класу та дружньої функції-оператора.
3. Чи губить оператор під час перевантаження своє вихідне призначення?
4. Чи можна змінити пріоритет перевантаженого оператора?
5. Чи можна змінювати кількість операндів у разі перевантаження операторів?
6. Чим дія дружньої функції-оператора відрізняються від дії оператора-функції члена класу?

ТЕМА 6. УСПАДКУВАННЯ

Зміст:

Види спадкування
Просте успадкування
Правила успадкування різних методів
Правила для деструкторів при успадкуванні
Множинне успадковування
Альтернатива успадковування

При великій кількості ніяк не пов’язаних класів управляти ними стає неможливим. Спадкування дозволяє впоратися з цією проблемою шляхом упорядкування та ранжирування класів, тобто об’єднання спільних для декількох класів властивостей в одному класі і використання його в якості базового.

Механізм успадкування класів дозволяє будувати ієрархії, в яких похідні класи отримують елементи батьківських, або базових, класів і можуть доповнювати їх або змінювати їх властивості.

Класи, що знаходяться ближче до початку ієрархії, об’єднують в собі найбільш загальні риси для всіх нижчих класів. У міру просування вниз по ієрархії класи набувають все більше конкретних рис. Множинне успадкування дозволяє одного класу мати властивості двох і більше батьківських класів.

Види спадкування

При описі класу в його заголовку перераховуються всі класи, які є для нього базовими. Можливість звернення до елементів цих класів регулюється за допомогою модифікаторів успадкування private, protected і public.

Якщо базових класів кілька, то вони перераховуються через кому. Перед кожним може стояти свій модифікатор успадкування. За замовчуванням для класів встановлений модифікатор private, а для структур - public.

Якщо заданий модифікатор успадкування public, воно називається відкритим. Використання модифікатора робить спадкування захищеним, а модифікатора private - закритим. Залежно від виду спадкування класи поводяться по різному. Клас може успадковувати від структури, і навпаки.

Для будь-якого елементу класу може також використовуватися специфікатор protected, який для одиночних класів, що не входять в ієрархію, рівносильний private. Різниця між ними проявляється при спадкуванні. Можливі поєднання модифікаторів і специфікаторів доступу наведені в таблиці нижче.

Як видно з таблиці нижче, елементи базового класу в похідному класі недоступні незалежно від ключа. Звернення до них може здійснюватися лише через методи базового класу.

Модифікатор успадкування	Специфікатор базового класу	Доступ в похідному класі
private	private	немає
	protected	private
	public	private
protected	private	немає
	protected	protected
	public	protected
public	private	немає
	protected	protected
	public	public

Елементи protected при спадкуванні з ключем private стають в похідному класі private, в інших випадках права доступу до них не змінюються.

Доступ до елементів public при спадкуванні стає відповідним ключу доступу.

Якщо базовий клас успадковується з ключем можна вибірково зробити деякі його елементи доступними в похідному класі, оголосивши їх в секції похідного класу за допомогою операції доступу до області видимості:

```
class Base {...
public: void f ();
};
class Derived: private Base {...
public: using Base :: f;
};
```

Просте успадкування

Простим називається успадкування, при якому похідний клас має одного з батьків. Для різних елементів класу існують різні правила успадкування.

У наведеному прикладі базовий клас Circle розширюється класом CircleColor. Клас CircleColor є підвидом класу Circle і додає до нього поле кольору.

У класі Circle реалізовано наступні елементи:

- внутрішні приховані (private) поля x, y, r;
- конструктор Circle() з 3 параметрами, які заповнюють значення внутрішніх полів;
- методи доступу GetXYR(), SetXYR() до полів класу;
- функція Area() обчислення площі круга.

У похідному класі CircleColor реалізовано поля та методи які доповнюють клас Color:

- внутрішнє приховане поле класу color;
- конструктор CircleColor() з 4 параметрами, який звертається до конструктора класу Color;
- методи GetColor(), SetColor() для доступу до прихованої змінної color.

Демонстраційний текст програми на C++ типу ConsoleApplication наступний:

```
#include <iostream>
using namespace std;

// C++. Демонстрація відношення is-a
// Клас, який реалізує коло - базовий клас
class Circle
{
 // 1. Внутрішні приховані поля класу
private:
 double x, y; // Координати центру кола
 double r; // радіус кола

public:
 // 2. Конструктор класу
 Circle(double x, double y, double r)
 {
  this->x = x;
  this->y = y;
  if (r > 0) this->r = r;
  else
     this->r = 1.0;
 }

 // 3. Методи доступу - геттери, сеттери
 void GetXYR(double& x, double& y, double& r)
 {
  x = this->x;
  y = this->y;
  r = this->r;
 }

 void SetXYR(double x, double y, double r)
 {
  this->x = x;
  this->y = y;

  // відкорегувати радіус якщо потрібно
  if (r > 0)
    this->r = r;
  else
    this->r = 1.0;
 }
```

```
  // 4. Функція Area() - площа кола
  double Area()
  {
    const double Pi = 3.141592;
    return Pi * r * r;
  }
};

// Клас, який розширює можливості класу Circle - додає колір кола
class CircleColor : public Circle
{
  // 1. Внутрішні змінні класу
private:
  unsigned int color = 0; // колір кола

public:
  // 2. Конструктор
  CircleColor(double x, double y, double r, unsigned int color) :Circle(x, y, r)
  {
    this->color = color;
  }

  // 3. Методи доступу
  unsigned int GetColor()
  {
    return color;
  }

  void SetColor(unsigned int color)
  {
    this->color = color;
  }
};

void main()
{
  // Демонстрація відношення між класами is-a
  // 1. Клас Circle
  // 1.1. Створити екземпляр класу Circle
  Circle cr(1, 3, 2);

  // 1.2. Викликати метод Area() класу Circle
  double area = cr.Area();
```

```
  cout << «The instance of class Color:» << endl;
  cout << «area = « << area << endl;

  // 1.3. Взяти значення полів екземпляру cr
  double x, y, r;
  cr.GetXYR(x, y, r);

  // 1.4. Вивести значення полів на екран
  cout << «x = « << x << «, y = « << y << «, radius = « << r << endl;

  // 2. Екземпляр класу CircleColor
  // 2.1. Створити екземпляр класу CircleColor
  CircleColor crCol(2, 4, 6, 1);

  // 2.2. Отримати значення полів екземпляру crCol
  // 2.2.1. Взяти значення кольору     з методу класу CircleColor
  unsigned int col = crCol.GetColor();

  // 2.2.2. Взяти значення x, y, r з методу базового класу Color
  crCol.GetXYR(x, y, r);

  // 2.3. Вивести отримані значення на екран
  cout << «The instance of class ColorCircle: « << endl;
  cout << «color = « << col << endl;
  cout << «x = « << x << endl;
  cout << «y = « << y << endl;
  cout << «r = « << r << endl;

  // 2.4. Викликати метод Area() базового класу Circle
  cout << «area = « << crCol.Area() << endl;
}
```

Результат виконання програми

```
The instance of class Color:
area = 12.5664
x = 1, y = 3,   radius = 2
The instance of class ColorCircle:
color = 1
x = 2
y = 4
r = 6
area = 113.097
```

Правила успадкування різних методів

Конструктори не успадковуються, тому похідний клас повинен мати власні конструктори.

Якщо в конструкторі похідного класу явний виклик конструктора базового класу відсутня, автоматично викликається конструктор базового класу за замовчуванням (тобто той, який можна викликати без параметрів).

Для ієрархії, що складається з декількох рівнів, конструктори базових класів викликаються починаючи з самого верхнього рівня. Після цього виконуються конструктори тих елементів класу, які є об'єктами, в порядку їх оголошення в класі, а потім виповнюється конструктор класу.

У разі декількох базових класів їх конструктори викликаються в порядку оголошення.

Якщо конструктор базового класу вимагає вказівки параметрів, він повинен бути явним чином викликаний в конструкторі похідного класу в списку ініціалізації (це продемонстровано в трьох останніх конструкторах).

Не успадковується і операція присвоювання, тому її також потрібно явно визначити в класі. Зверніть увагу на запис функції-операції: в її тілі застосований явний виклик функції-операції присвоювання з базового класу. Щоб краще уявити собі синтаксис виклику, ключове слово operator разом зі знаком операції можна інтерпретувати як ім'я функції-операції.

Виклик функцій базового класу краще копіювання фрагментів коду з функцій базового класу в функції похідного. Крім скорочення обсягу коду, цим досягається спрощення модифікації програми: зміни потрібно вносити тільки в одну точку програми, що скорочує кількість можливих помилок.

Правила для деструкторів при успадкуванні

Деструктори не успадковуються, і якщо програміст не описав в похідному класі деструктор, він формується за замовчуванням і викликає деструктори всіх базових класів.

На відміну від конструкторів, при написанні деструктора похідного класу в ньому не потрібно явно викликати деструктори базових класів, оскільки це буде зроблено автоматично.

Для ієрархії класів, що складається з декількох рівнів, деструктори викликаються в порядку, строго зворотному виклику конструкторів: спочатку викликається деструктор класу, потім - деструктори елементів класу, а потім деструкція базового класу.

Поля, успадковані з класу, недоступні функцій похідного класу, оскільки вони визначені в базовому класі як private. Якщо функції потрібно працювати з цими полями, можна або описати їх в базовому класі як protected, або звертатися до них за допомогою функцій з базового класу, або явно перевизначити їх.

Поля, які додаються в нащадка можуть збігатися і по імені, і за типом з полями базового класу. При цьому поле предка буде приховано.

Статичні поля, оголошені в базовому класі, успадковуються звичайним

чином. Всі об’єкти базового класу і всіх його спадкоємців поділяють єдину копію статичних полів базового класу.

Похідний клас може не тільки доповнювати, але і коректувати поведінку базового класу. Доступ до перевизначення методу базового класу для похідного класу виконується через уточнене за допомогою операції доступу до області видимості ім’я.

Клас-нащадок успадковує всі методи базового класу, крім конструкторів, деструктора і операції привласнення. Чи не успадковуються ні дружні функції, ні дружні відносини класів.

У класі-спадкоємця можна визначати нові методи. У них дозволяє здійснювати будь-які доступні методи базового класу. Якщо ім’я методу в спадкоємця збігається з ім’ям методу базового класу, то метод похідного класу приховує всі методи базового класу з таким ім’ям. При цьому прототипи методів можуть не збігатися. Якщо в методі-спадкоємця потрібно викликати однойменний метод батьківського класу, потрібно задати його з префіксом класу. Це ж стосується і статичних методів.

Множинне успадковування

Множинне успадкування означає, що клас має декілька базових класів. При цьому, якщо в базових класах є однойменні елементи, може статися конфлікт ідентифікаторів, який усувається за допомогою операції доступу до області видимості.

Якщо у базових класів є загальний предок, це призведе до того, що похідний від цих базових клас успадкує два примірника полів предка, що найчастіше є небажаним. Щоб уникнути такої ситуації, потрібно при спадкуванні загального предка визначити його як віртуальний клас.

Альтернатива успадковування

Альтернативою успадкування при проектуванні класів є вкладення, коли один клас включає в себе поля, які є класами або покажчиками на них. Наприклад, якщо є клас «двигун», а потрібно описати клас «літак», логічно зробити двигун полем цього класу, а не його предком.

Вид вкладення, коли в класі описано поле об’єктного типу, називають композицією. Якщо в класі описаний покажчик на об’єкт іншого класу, це зазвичай називають агрегацией. При композиції час життя всіх об’єктів (і доступних, і його полів) однаково. Агрегація представляє собою більш слабкий зв’язок між об’єктами, тому що об’єкти, на які посилаються поля-покажчики, можуть з’являтися і зникати протягом життя вмісту їх об’єкта, крім того, один і той же покажчик може посилатися на об’єкти різних класів в межах однієї ієрархії. Поле-покажчик може також посилатися на один об’єкт, а на невизначену кількість об’єктів, наприклад бути дороговказом на початок лінійного списку.

У даному скороченому коді демонструється агрегація на прикладі класів Triangle, Circle, Figures. Клас Figuresможе містити різну кількість різних фігур

(навіть 0). При можливості до класу Figures може бути додано масиви інших фігур, наприклад, Rectangle (прямокутник). У будь-якому випадку клас Figures буде повноцінним функціонально, отже, це є агрегація.

```
// Клас трикутник
class Triangle
{
  // Методи та поля класу Triangle
  // ...
};

// Клас, що реалізує коло
class Circle
{
  // Методи та поля класу Circle
  // ...
};

// Клас, що реалізує різні геометричні фігури.
// Використовується тип відношення - агрегація.
class Figures
{
  Triangle tr[10]; // масив трикутників
  unsigned int n_tr; // к-сть трикутників у масиві tr
  Circle cr[10]; // масив кіл
  unsigned int n_cr; // к-сть кіл у масиві cr

  // Інші поля та методи класу
  // ...
};
```

У наступному прикладі продемонстровано композицію для класів Vehicle, Wheel, Car. В автомобіль (Car) входять двигун (клас Vehicle) та колесо (Wheel), які є його складовою частиною – це є композиція (об’єднання).

```
// Клас Двигун
class Vehicle
{
  // Поля та методи класу
  // ...
};

// Клас Колесо
```

```
class Wheel
{
  // Поля та методи класу
  // ...
};

// Клас Автомобіль - містить обов’язкові елементи,
// які є складовими Автомобіля
class Car
{
  // Екземпляри обов’язкових класів,
  // що є частиною даного класу - це є композиція
private:
  Vehicle veh; // автомобіль містить двигун (обов’язково)
  Wheel whl[4]; // колеса - 4 штуки (обов’язково)

  // Поля та методи класу
  // ...
};
```

Контрольні завдання та запитання

1. Поясніть принцип ООП — успадкування. Наведіть приклади.
2. Які ви знаєте специфікатори доступу?
3. Що відбувається з відкритими та закритими членами базового класу, якщо базовий клас успадковується як відкритий похідним?
4. Що відбувається з закритими та відкритими членами базового класу, якщо базовий клас успадковується як закритий похідним?
5. Поясніть, навіщо потрібна категорія захищеності protected? Розгляньте два випадки: коли protected використовується всередині описання класу та під час успадкування.
6. Якщо один клас успадковується іншим, яким порядок виклику конструкторів та деструкторів? Наведіть приклади.
7. Що таке множинне успадкування? Які ви знаєте можливі варіанти множинного успадкування?

ТЕМА 7. ВІРТУАЛЬНІ ФУНКЦІЇ ТА ШАБЛОНИ

Зміст:

Віртуальні функції
Чиста віртуальна функція
Механізм пізнього зв'язування
Віртуальний деструктор
Абстрактні класи
Шаблони функцій
Створення простого шаблону функції
Шаблони класів

Віртуальні функції

Віртуальні функції - спеціальний вид функцій-членів класу. Віртуальна функція відрізняється про звичайну функції тим, що для нормальної функції зв'язування виклику функції з її визначенням здійснюється на етапі компіляції. Для віртуальних функцій це відбувається під час виконання програми.

Для оголошення віртуальної функції використовується ключове слово virtual. Функція-член класу може бути оголошена як віртуальна, якщо

- клас, що містить віртуальну функцію, базовий в ієрархії породження;
- реалізація функції залежить від класу і буде різною в кожному породженому класі.

Віртуальна функція - це функція, яка визначається в базовому класі, а будь-який породжений клас може її перевизначити. Віртуальна функція викликається тільки через покажчик або посилання на базовий клас.

Визначення того, який екземпляр віртуальної функції викликається за висловом виклику функції, залежить від класу об'єкта, що адресується покажчиком або посиланням, і здійснюється під час виконання програми. Цей механізм називається динамічним (пізнім) зв'язуванням або дозволом типів під час виконання.

Покажчик на базовий клас може вказувати або на об'єкт базового класу, або на об'єкт породженого класу. Вибір функції-члена залежить від того, на об'єкт якого класу при виконанні програми вказує покажчик, але не від типу покажчика. При відсутності члена породженого класу за замовчуванням використовується віртуальна функція базового класу.

```
# include <iostream>
using namespace std;
class X
{
protected:
  int i;
public:
  void seti (int c) {i = c; }
```

```
  virtual void print () {cout << endl << «class X:» << i; }
};
class Y: public X // спадкування
{
public:
  void print () {cout << endl << «class Y:» << i; } // перевизначення базової функції
};
int main ()
{
  X x;
  X * px = & x; // Покажчик на базовий клас
  Y y;
  x.seti (10);
  y.seti (15);
  px-> print (); // клас X: 10
  px = & y;
  px-> print (); // клас Y: 15
  cin.get ();
  return 0;
}
```

У кожному разі виконується різна версія функції print (). Вибір динамічно залежить від об’єкта, на який посилається покажчик.

Якщо прибрати ключове слово virtual, то результат виконання буде вже іншим, тому що зв’язування функцій буде відбуватися на етапі компіляції:

Віртуальною може бути тільки нестатична функція-член класу. Для породженого класу функція автоматично стає віртуальною, тому ключове слово virtual можна опустити.

Приклад: вибір віртуальної функції

```
# include <iostream>
using namespace std;
class figure
{
protected:
  double x, y;
public:
  figure (double a = 0, double b = 0) {x = a; y = b; }
  virtual double area () {return (0); } // за замовчуванням
};
class rectangle: public figure
{
```

```
public:
  rectangle (double a = 0, double b = 0): figure (a, b) {};
  double area () {return (x * y); }
};
class circle: public figure
{
public:
  circle (double a = 0): figure (a, 0) {};
  double area () {return (3.1415 * x * x); }
};
int main ()
{
  figure * f [2];
  rectangle rect (3, 4);
  circle cir (2);
  double total = 0;
  f [0] = & rect;
  f [1] = & cir;
  total = f [1] -> area ();
  cout << total << endl;
  total + = f [0] -> area ();
  cout << total << endl;
  cin.get ();
  return 0;
}
```

Правила використання віртуальних методів:

Якщо в базовому класі метод визначений як віртуальний, метод, визначений в похідному класі з тим же ім’ям і набором параметрів, автоматично стає віртуальним, а з відмінним набором параметрів - звичайним.

Віртуальні методи успадковуються, тобто перевизначати їх в похідному класі потрібно тільки при необхідності задати відрізняються дії. Права доступу при перевизначенні змінити не можна.

Якщо віртуальний метод перевизначений у похідному класі, об’єкти цього класу можуть отримати доступ до методу базового класу за допомогою операції доступу до області видимості.

Віртуальний метод не може оголошуватися з модифікатором static, але може бути оголошений як дружня функція.

Якщо похідний клас містить віртуальні методи, вони повинні бути визначені в базовому класі хоча б як чисто віртуальні.

Чиста віртуальна функція

Базовий клас ієрархії типу зазвичай містить ряд віртуальних функцій, які забезпечують динамічну типізацію. Часто в самому базовому класі самі віртуальні функції фіктивні і мають порожнє тіло. Певне значення їм надається лише в породжених класах. Такі функції називаються чистими віртуальними функціями.

Чиста віртуальна функція - це метод класу, тіло якого не визначено.

У базовому класі така функція записується в такий спосіб:

```
virtual ПрототипФункції = 0;
```

Наприклад

```
virtual void func () = 0;
```

Чисті віртуальні функції використовуються для того, щоб відкласти рішення задачі про реалізацію функції на більш пізній термін. У термінології ООП це називається відстроченим методом. Клас, що має принаймні одну чисту віртуальну функцію, називається абстрактним базовим класом. Для ієрархії типу корисно мати абстрактний базовий клас. Він містить загальні властивості ієрархії типу, але кожен породжений клас реалізує ці властивості по-своєму.

Механізм пізнього зв'язування

Для кожного класу (не об'єкта!), що містить хоча б один віртуальний метод, компілятор створює таблицю віртуальних методів (vtbl), в якій для кожного віртуального методу записано його адресу в пам'яті. Адреси методів містяться в таблиці в порядку їх опису в класах. Адреса будь-якого віртуального методу має в vtbl одне і те ж зміщення для кожного класу в межах ієрархії.

Кожен об'єкт містить приховане додаткове поле посилання на vtbl, зване vptr. Воно заповнюється конструктором при створенні об'єкта (для цього компілятор додає в початок тіла конструктора відповідні інструкції).

На етапі компіляції посилання на віртуальні методи замінюються на звернення до через vptr об'єкта, а на етапі виконання в момент звернення до методу його адресу вибирається з таблиці. Таким чином, виклик віртуального методу, на відміну від звичайних методів і функцій, виконується через додатковий етап отримання адреси методу з таблиці. Це дещо уповільнює виконання програми, тому без необхідності робити методи віртуальними сенсу не має.

Рекомендується робити віртуальними деструктори для того, щоб гарантувати правильне звільнення пам'яті з-під динамічного об'єкта, оскільки в будь-який момент часу буде обраний деструктор, відповідний фактичному типу об'єкта.

Чіткого правила, за яким метод слід робити віртуальним, не існує. Можна тільки дати рекомендацію оголошувати віртуальними методи, для яких є ймовірність, що вони будуть перевизначені в похідних класах. Методи, які у всій

ієрархії залишаться незмінними або ті, якими похідні класи користуватися не будуть, робити віртуальними немає сенсу. З іншого боку, при проектуванні ієрархії не завжди можна передбачити, яким чином будуть розширюватися базові класи, особливо при проектуванні бібліотек класів, а оголошення методу віртуальним забезпечує гнучкість і можливість розширення.

Для пояснення останньої тези уявімо собі, що виклик методу draw здійснюється з методу переміщення об'єкта. Якщо текст методу переміщення не залежить від типу переміщуваного об'єкта (оскільки принцип переміщення всіх об'єктів однаковий, а для відтворення викликається конкретний метод), перевизначати цей метод в похідних класах немає необхідності, і він може бути описаний як невіртуальний. Якщо метод віртуальний, метод переміщення зможе без перекомпіляції працювати з об'єктами будь-яких похідних класів - навіть тих, про яких при його написанні нічого відомо не було.

Віртуальний механізм працює тільки при використанні покажчиків або посилань на об'єкти. Об'єкт, визначений через покажчик або посилання і містить віртуальні методи, називається поліморфним. В даному випадку поліморфізм полягає в тому, що за допомогою одного і того ж звернення до методу виконуються різні дії в залежності від типу, на який посилається вказівник в кожен момент часу.

Віртуальний деструктор

У мові програмування С ++ деструктор поліморфного базового класу повинен оголошуватися віртуальним. Тільки так забезпечується коректне руйнування об'єкта похідного класу через покажчик на відповідний базовий клас.

Розглянемо наступний приклад:

```
#include <iostream>
using namespace std;
// Допоміжний клас
class Об'єкт
{
public:
Object () {cout << «Object :: ctor ()» << endl; }
~ Object () {cout << «Object :: dtor ()» << endl; }
};
// Базовий клас
class Base
{
public:
Base () {cout << «Base :: ctor ()» << endl; }
virtual ~ Base () {cout << «Base :: dtor ()» << endl; }
virtual void print () = 0;
};
```

```
// Похідний клас
class Derived: public Base
{
public:
Derived () {cout << «Derived :: ctor ()» << endl; }
~ Derived () {cout << «Derived :: dtor ()» << endl; }
void print () {}
Object obj;
};
int main ()
{
Base * p = new Derived;
delete p;
return 0;
}
```

У функції main вказівником на базовий клас присвоюється адреса динамічно створюваного об'єкта похідного класу Derived. Потім через цей покажчик об'єкт руйнується. При цьому наявність віртуального деструктора базового класу забезпечує виклики деструкторів всіх класів в очікуваному порядку, а саме, в порядку, зворотному викликам конструкторів відповідних класів.

Висновок програми з використанням віртуального деструктора в базовому класі буде наступним:

```
Base :: ctor ()
Object :: ctor ()
Derived :: ctor ()
Derived :: dtor ()
Object :: dtor ()
Base :: dtor ()
```

Знищення об'єкта похідного класу через покажчик на базовий клас з невіртуальну деструктором дає невизначений результат. На практиці це виражається в тому, що буде зруйнована тільки частина об'єкта, відповідна базового класу. Якщо в коді вище прибрати ключове слово virtual перед деструктором базового класу, то висновок програми буде вже іншим. Зверніть увагу, що член даних obj класу Derived також не руйнується.

```
Base :: ctor ()
Object :: ctor ()
Derived :: ctor ()
Base :: dtor ()
```

Коли ж слід оголошувати деструктор віртуальним? Існує правило - якщо базовий клас призначений для поліморфного використання, то його деструктор повинен оголошуватися віртуальним. Для реалізації механізму віртуальних функцій кожен об’єкт класу зберігає покажчик на таблицю віртуальних функцій vptr, що збільшує його загальний розмір. Зазвичай, при оголошенні віртуального деструктора такий клас вже має віртуальні функції, і збільшення розміру відповідного об’єкта не відбувається.

Якщо ж базовий клас не призначений для поліморфного використання (не містить віртуальних функцій), то його деструктор не повинен оголошуватися віртуальним.

Абстрактні класи

Клас, що містить хоча б один чисто віртуальний метод, називається абстрактним. Абстрактні класи призначені для представлення спільних понять, які передбачається конкретизувати в похідних класах. Абстрактний клас може використовуватися тільки в якості базового для інших класів - об’єкти абстрактного класу створювати не можна, оскільки прямий або непрямий виклик чисто віртуального методу призводить до помилки при виконанні.

При визначенні абстрактного класу необхідно мати на увазі наступне:

Абстрактний клас не можна використовувати при явному приведення типів, для опису типу параметра і типу що повертається функцією значення.

Допускається оголошувати покажчики і посилання на абстрактний клас, якщо при ініціалізації не потрібно створювати тимчасовий об’єкт.

Якщо клас, похідний від абстрактного, не визначає все чисто віртуальні функції, він також є абстрактним.

Таким чином, можна створити функцію, параметром якої є покажчик на абстрактний клас. На місце цього параметра при виконанні програми може передаватися покажчик на об’єкт будь-якого похідного класу. Це дозволяє створювати поліморфні функції, що працюють з об’єктом будь-якого типу в межах однієї ієрархії.

Шаблони функцій

При створенні функцій іноді виникають ситуації, коли дві функції виконують однакову обробку, але працюють з різними типами даних (наприклад, одна використовує параметри типу int, а інша типу float). Ви вже знаєте, що за допомогою механізму перевантаження функцій можна використовувати одне і те ж ім’я для функцій, що виконують різні дії і мають різні типи параметрів. Однак, якщо функції повертають значення різних типів, вам слід використовувати для них унікальні імена. Припустимо, наприклад, що у вас є функція з ім’ям max, яка повертає максимальне з двох цілих значень. Якщо пізніше вам буде потрібно подібна функція, яка повертає максимальне з двох значень з плаваючою точкою, вам слід визначити іншу функцію, наприклад fmax.

Шаблон визначає набір операторів, за допомогою яких ваші програми

пізніше можуть створити кілька функцій.

Програми часто використовують шаблони функцій для швидкого визначення декількох функцій, які за допомогою однакових операторів працюють з параметрами різних типів або повертають різні типи значень.

Шаблони функцій мають специфічні імена, які відповідають імені функції, використовуваному вами в програмі.

Після того як ваша програма визначила шаблон функції, вона в подальшому може створити конкретну функцію, використовуючи цей шаблон для завдання прототипу, який включає ім'я цього шаблону, що повертається функцією значення і типи параметрів.

В процесі компіляції компілятор C ++ буде створювати у вашій програмі функції з використанням типів, зазначених в прототипах функцій, які посилаються на ім'я шаблону.

Створення простого шаблону функції

За допомогою такого шаблону ваші програми в подальшому можуть визначити конкретні функції з необхідними типами. Наприклад, нижче визначено шаблон для функції з ім'ям max, яка повертає більше з двох значень:

```
template <class T> T max (T a, T b)
{
if (a> b) return (a);
else return (b);
}
```

Буква T даному випадку являє собою загальний тип шаблону. Після визначення шаблону всередині вашої програми ви оголошуєте прототипи функцій для кожного необхідного вам типу. У разі шаблону max наступні прототипи створюють функції типу float і int.

```
float max (float, float);
int max (int, int);
```

Коли компілятор C ++ зустріне ці прототипи, то при побудові функції він замінить тип шаблону зазначеним вами типом. У випадку з типом функція після заміни прийме наступний вигляд:

```
template <class T> T max (T a, T b)
{
if (a> b) return (a);
else return (b);
}
float max (float a, float b)
```

```
{
if (a> b) return (a);
else return (b);
}
```

У міру того як ваші програми стають більш складними, можливі ситуації, коли вам будуть потрібні подібні функції, які виконують одні й ті ж операції, але з різними типами даних. Шаблон функції дозволяє вашим програмам визначати загальну, або тіпонезавісімую, функцію. Коли програмі потрібно використовувати функцію для певного типу, наприклад int або float, вона вказує прототип функції, який використовує ім'я шаблону функції і типи значення, що повертається і параметрів. В процесі компіляції C ++ створить відповідну функцію. Створюючи шаблони, ви зменшуєте кількість функцій, які повинні кодувати самостійно, а ваші програми можуть використовувати один і той же ім'я для функцій, що виконують певну операцію, незалежно від що повертається функцією значення і типів параметрів.

Шаблони класів

Шаблон класу дозволяє задати клас, параметризовані типом даних. Передача класу різних типів даних в якості параметра створює сімейство споріднених класів. Найбільш широке застосування шаблони знаходять при створенні контейнерних класів. Контейнерним називається клас, який призначений для зберігання будь-яким чином організованих даних і роботи з ними. Перевага використання шаблонів полягає в тому, що як тільки алгоритм роботи з даними визначено і налагоджений, він може застосовуватися до будь-яких типів даних без переписування коду.

Шаблон класу починається з ключового слова template. У кутових дужках записують параметри шаблону. При використанні шаблону на місце цих параметрів шаблоном передаються аргументи: типи і константи, перераховані через кому.

```
template <опис_параметрів_шаблона> class ім'я {/ * визначення класу * /};
```

Типи можуть бути як стандартними, так і певними користувачем. Для їх опису в списку параметрів використовується ключове слово class. У найпростішому випадку одного параметра це виглядає як <class T>. Тут T є параметром-типом. Ім'я параметра може бути будь-яким, але прийнято починати його з префікса T. Усередині класу-шаблону параметр може з'являтися в тих місцях, де дозволяється вказувати конкретний тип.

Опис параметрів шаблону в заголовку функції має відповідати шаблоном класу.

Локальні класи не можуть мати шаблони в якості своїх елементів.

Шаблони методів не можуть бути віртуальними.

Шаблони класів можуть містити статичні елементи, дружні функції і класи.

Шаблони можуть бути похідними як від шаблонів, так і від звичайних класів, а також бути базовими і для шаблонів, і для звичайних класів.

Всередині шаблону не можна визначати friend-шаблони.

Контрольні завдання та запитання

1. Що таке віртуальна функція?
2. Які функції не можуть бути віртуальними?
3. Яка відмінність між віртуальною та перевантаженою функціями?
4. Що таке чисто віртуальна функція? Які її відмінності від звичайної віртуальної функції?
5. Що таке абстрактний клас? Що таке поліморфний клас?
6. Що таке родова функція?
7. Яка загальна форма родової функції?
8. Чи може родова функція мати кілька родових аргументів?
9. Яка загальна форма родової функції з кількома родовими аргументами?
10. Яка відмінність між родовою та перевантаженою функціями?
11. Що таке родовий (параметризований клас)?
12. Яка загальна форма родового класу?
13. Яка загальна форма родового класу з кількома родовими типами даних?
14. Що таке контейнери? Наведіть приклади.
15. Які переваги використання родових класів?
16. Яка структура зв’язного списку? Яке його застосування?
17. Як організовано чергу? Які застосування черги ви знаєте?
18. Як організовано стек? Які застосування стеку ви знаєте?

ТЕМА 8. СИСТЕМА ВВЕДЕННЯ-ВИВЕДЕННЯ

Зміст:

Базові положення системи введення-виведення C++
Форматне введення-виведення даних
Використання функцій width(), precision() і fill()
Маніпулятори введення-виведення
Файлове введення-виведення
Створення власних функцій вставки
Створення власних функцій вилучення
Створення власних маніпуляторів

Базові положення системи введення-виведення C++

Система введення-виведення C++ діє через потоки (streams).

Потік — це логічний пристрій, який видає та приймає інформацію. Потік зв'язаний з фізичним пристроєм за допомогою системи введення-виведення C++.

Коли починається програма мовою C++, автоматично відкриваються чотири потоки (інформацію про них наведено в табл. 2).

У C++ система введення-виведення підтримується заголовним файлом iostream.h. У цьому файлі для підтримання введення-виведення задано ієрархію класів. Клас нижнього рівня введення-виведення називається streambuf. Цей клас забезпечує базові дії з уведення-виведення та використовується переважно як базовий для інших класів. Ще один клас в ієрархії — ios. Цей клас забезпечує форматування, контроль помилок та інформацію про стан потоків уведення-виведення.

Таблиця 1

Стандартні маніпулятори введення-виведення

Маніпулятор	Призначення	Уведення-виведення
dec	Виведення числових даних в десятковій системі числення	Виведення
endl	Виведення символа нового рядка і флешування	Виведення
ends	Виведення нуля (NULL)	Виведення
flush	Флешування	Виведення
hex	Виведення числових даних у шістнадцятковій системі числення	Виведення
oct	Виведення числових даних у вісімковій системі числення	Виведення
resetiosflags(long f)	Скидає прапори, задані f	Уведення-виведення
setbase(int основа)	Установлює основу системи числення	Виведення
setfill(int ch)	Установлює символ заповнення ch	Виведення
setiosflags(long f)	Установлює прапори, задані f	Уведення-виведення
setprecision(int p)	Задає кількість цифр після десяткової точки, що дорівнює p	Виведення

setw(int w)	Задає w позицій ширини поля	Виведення
ws	Пропуск початкових пробілів	Уведення

Таблиця 2

Стандартні потоки у C++

Потік	Значення	Пристрій за замовчуванням
cin	Стандартний ввод	Клавіатура
cout	Стандартний виведення	Екран
cerr	Стандартна помилка	Екран
clog	Буферизована версія cerr	Екран

Ios використовується як базовий для трьох класів: istream, ostream та iostream. Ці класи застосовуються для створення потоків, сумісних відповідно з уведенням, виведенням та введенням-виведенням. Клас ios має багато функцій та змінних – членів класу, які керують та контролюють основну роботу потоку. Коли файл iostram.h включено в програму, можна отримати доступ до класу ios.

Форматне введення-виведення даних

Інформацію можна виводити в широкому діапазоні форм. Кожний потік C++ зв'язаний з набором прапорів формату, які задають формат відображення даних.

Для встановлення прапора формату користуються функцією *setf()*. Ця функція є членом класу *ios*. Її типова форма:

```
long setf(long прапори);
```

Функція *setf()* повертає попередні установлення прапорів формату і встановлює задані прапори. Наприклад, для встановлення прапора *showpos* можна скористатися оператором:

```
потік.setf(ios::showpos);
```

Тут потік — це той потік, на який можна впливати; *showpos* — це константа, що входить в *enumeration* всередині класу *ios*. Тому, щоб повідомити компілятору про це, необхідно поставити перед *showpos* ім'я класу і операцію розширення діапазону бачення.

Замість кількох викликів *setf(),* можна встановлювати більш одного прапора за один виклик. Для того щоб об'єднати необхідні прапори, використовується операція OR. Наприклад,

```
cout.setf(ios::showbase | ios::hex);
```

Для скинення одного або декількох прапорів формату, використовується функція *unsetf()*:

```
long unsetf(long прапори);
```

В *ios* також внесено функцію-член *flags()*, яка повертає поточний стан прапорів формату в змінній типу *long*.

Приклад 1. Робота функцій *setf()* і *unsetf()*:

```
#include <iostream.h> int main() {
cout.setf(ios::uppercase|ios::showbase|ios::hex); cout << 88 << '\n';
cout.unsetf(ios::uppercase); cout << 88 << '\n';
return 0;
}
```

У програмі спочатку встановлюються прапори *uppercase, showbase* і *hex*. Потім виводиться число 88 з використанням наукової нотації. В цьому випадку шістнадцяткове "x" виводиться у верхньому регістрі. Далі *unsetf()* скидає прапор *uppercase* і знову виводиться шістнадцяткове 88. Тепер "x" буде у нижньому регістрі.

Використання функцій width(), precision() і fill()

Крім прапорів формату, існують три функції-члени, що визначені в класі *ios*, які визначають параметри формату: ширину поля, точність і символ заповнення. Це, відповідно, функції *width(), precision()* і *fill()*.

За замовчуванням під час виведення будь-якого значення воно займає кількість позицій, що відповідає кількості символів, що виводяться. Проте, використовуючи функцію *width()*, можна задати мінімальну ширину поля. Прототип функції такий:

```
int width(int w);
```

Тут *w* — ширина поля, а функція повертає попередню ширину поля.

Після встановлення мінімальної ширини поля, якщо значення, що виводиться, потребує меншої ширини поля, то його решта заповнюється поточним символом заповнення (за замовчуванням пробілом). Проте, якщо розмір значення, що виводиться, перевищує мінімальну ширину поля, буде зайнято стільки символів, скільки треба.

За замовчуванням при виведенні значень з плаваючою точкою, після десяткової точки ставиться шість цифр. Використовуючи функцію *precision()*, це значення можна змінити. Прототип функції такий:

```
int precision(int p);
```

Тут *p* – це точність (кількість цифр, що виводяться після коми), сама функція повертає попередню точність.

За замовчуванням при заповненні поля використовуються пробіли. Проте можна змінити символ заповнення, використовуючи функцію *fill()*. Її прототип такий:

```
char fill(char ch);
```

Після виклику функції *fill()* змінна *ch* становиться символом заповнення, а функція повертає попередній символ заповнення.

Маніпулятори введення-виведення

У C++ є інший спосіб форматування інформації з використанням системи введення-виведення C++. За цього способу застосовуються спеціальні функції – маніпулятори введення-виведення (i/o manipulators). У деяких випадках маніпулятори більш зручні, ніж прапори формату та функції класу *ios*.

Маніпулятори введення-виведення — це спеціальні функції формату введення-виведення, які можуть міститися в тілі оператора введення-виведення; більшість з них діють аналогічно функціям-членам класу *ios*.

Наприклад:

```
cout<<oct<<100<<hex<<100; cout<<setw(10)<<100;
```

Перший оператор повідомляє *cout* про необхідність виведення цілих у вісімковій системі числення, потім виводиться значення 100 у вісімковій системі числення. Далі він повідомляє потоку про необхідність виведення цілих у шістнадцятковій системі числення і далі виводиться число 100 у шістнадцятковому форматі. У другому операторі встановлюється ширина поля 10, і потім знову виводиться 100 у шістнадцятковому форматі.

Маніпулятор введення-виведення впливає лише на потік, який є частиною виразу введення-виведення, що має маніпулятор. Маніпулятори введення-виведення не впливають на всі потоки, відкриті в поточний момент часу для використання.

Файлове введення-виведення

Для реалізації файлового введення-виведення необхідно включити в програму заголовний файл fstream.h. У ньому визначено декілька класів, включаючи ifstream, ofstream і fstream. Ці класи є похідними класів istream і ostream.

У C++ файл відкривається за допомогою його зв’язування з потоком. Є три види потоків: вхідний, вихідний і вхідний- вихідний. Перед тим, як відкрити файл, передусім треба створити потік. Для створення вхідного потоку необхідно оголосити потік класу *ifstream*, для створення вихідного — оголосити потік класу *ofstream*. Потоки, які реалізують і введення, і виведення, повинні оголошуватися як об’єкти класу *fstream*. Наприклад:

```
ifstream in; // введення
ofstream on; // виведення
fstream io; // введення і виведення
```

Після створення потоку, одним зі способів зв'язати його з файлом є функція *open()*. Ця функція є членом кожного з трьох початкових класів. Її прототип такий:

```
void open(char *filename, int mode, int access);
```

Тут *filename* — ім'я файлу, в який можна включити шлях. Величина *mode* задає параметри відкриття файлу. Параметр *access* задає права доступу до файлу. За замовчуванням значення *access* дорівнює *filebuf::openprot* і дорівнює 0÷644 для середовища UNIX, що означає звичайний клас. У середовищі DOS/WINDOWS, *access* відповідає кодам атрибутів файлів DOS/ WINDOWS.

Хоча використовувати функцію *open()* для відкриття файлу в цілому правильно, на практиці частіше це не робиться, оскільки у класів *ifstream, ofstream* і *fstream* є конструктори, які відкривають файл автоматично. Конструктори мають такі самі параметри, що задані за замовчуванням у функції *open()*.

Наприклад,

```
ifstream mystream("myfile");
//Відкриття файлу введення
```

Для закриття файлу використовується функція-член *close()*. Наприклад, щоб закрити файл, зв'язаний з потоком *mystream*, використовується оператор:

```
mystream.close();
```

Функція *close()* не має параметрів та значення, що повертається.

Використовуючи функцію-член *eof()*, можна визначити, чи був досягнутий кінець файлу під час уведення.

Прототип функції такий:

```
int eof();
```

Вона повертає ненульове значення в тому випадку, якщо досягнуто кінець файлу; в противному випадку функція повертає нуль.

Після відкриття файлу для введення-виведення інформації використовуються операції "<<" і " >>" і зв'язаний з файлом потік.

Створення власних функцій вставки

Однією з переваг використання операторів уведення- виведення C++ замість аналогічних функцій введення- виведення C є можливість перевантаження операторів уведення- виведення для створених класів.

У мові C++ виведення іноді називається **вставкою** (insertion), а оператор "<<" — оператором вставки. Зміст цього терміна полягає в тому, що операція виведення вставляє (inserts) інформацію в потік. Коли перевантажується "<<" для виведення, створюється функція вставки (inserter function) або (inserter).

У всіх функцій вставки така загальна форма:

```
ostream &operator<<(ostream &stream, ім'я_класу ob) {
… //Тіло функції вставки
return stream;
};
```

Перший параметр є посиланням на об'єкт типу *ostream*. Це означає, що *stream* має бути потоком виведення. Другий параметр отримує об'єкт для виведення, він також може бути посиланням, якщо це треба. Всередині функції вставки можна виконувати будь-яку процедуру, проте відповідно до хорошого стилю програмування рекомендується обмежувати роботу власної функції вставки тільки виведенням інформації в потік.

Функція вставки не може бути членом класу, для роботи з яким її задано (лівий операнд — потік, а не об'єкт класу). Тому, щоб мати доступ до закритих членів класу, функція вставки повинна бути дружньою класу.

Приклад 2. Створення дружньої функції вставки:

```
#include <iostream.h> class coord { int x, y;
public: coord() { x = 0; y= 0; } coord(int i, int j) { x = i; y = j; }
friend ostream &operator<<(ostream &stream, coord ob);
};
ostream &operator<<(ostream &stream, coord ob)
{ stream << ob.x << «, « << ob.y << '\n'; return stream; }
int main()
{
coord a(1, 1), b(10, 23); cout << a << b;
return 0;
}
```

Варто звернути увагу на те, що оператор уведення- виведення всередині функції виводить значення *x* і *y* в *stream*, який є довільним потоком, який передається у функцію.

Створення власних функцій вилучення

Оператор уведення “>>” можна перевантажувати так само, як і оператор виведення “<<”. У C++ оператор уведення “>>” називають оператором вилучення (extraction operator). Зміст цього терміна в тому, що під час уведення інформації з потоку вилучаються дані.

Загальна форма функції введення користувача така:

```
istream &operator>>(istream &stream, ім’я_класу &ob)
{
… //Тіло функції вилучення
return stream;
};
```

Функція вилучення повертає посилання на *istream*, який є потоком уведення. Перший параметр повинен бути посиланням на потік уведення, другий — це посилання на об’єкт, що отримує інформацію.

Функція вилучення не може бути функцією-членом. Усередині функції вилучення може виконуватися будь-яка операція, проте доцільно обмежити її роботу введенням інформації.

Приклад 3. Створення дружньої функції вилучення:

```
#include <iostream.h> class coord { int x, y;
public:
coord() { x = 0; y= 0; } coord(int i, int j) { x = i; y = j; }
friend ostream &operator<<(ostream &stream, coord ob); friend istream &operator>>(istream &stream, coord &ob);
};
ostream &operator<<(ostream &stream, coord ob) { stream << ob.x << «, « << ob.y << ‘\n’;
return stream;
}
istream &operator>>(istream &stream, coord &ob) { cout << «Введіть координати: «;
stream >> ob.x >> ob.y; return stream;
}
int main() {
coord a(1, 1), b(10, 23); cout << a << b;
cin >> a; cout << a; return 0;
}
```

Створення власних маніпуляторів

Як додаток до перевантаження операторів уведення- виведення, можна створити свою підсистему введення- виведення C++, визначивши власні

функції-маніпулятори. Маніпулятори користувача можуть допомогти створити будь- яку програму введення-виведення більш зрозумілішою і ефективнішою. Використання власних маніпуляторів важливе, оскільки можна об'єднувати декілька окремих дій з уведення- виведення в один маніпулятор, а також виконувати введення- виведення на нестандартному обладнанні.

Є два базові типи маніпуляторів: ті, які працюють з потоком введення, і ті, які працюють з потоком виведення. Є маніпулятори з параметрами і без них. Розглянемо маніпулятори без параметрів.

Усі маніпулятори без параметрів для виведення мають таку форму:

```
ostream &ім'я_маніпулятора(osream &stream) {
…//Код тіла
return stream;
};
```

Усі маніпулятори без параметрів для введення мають таку форму:

```
istream &ім'я_маніпулятора(isream &stream) {
…//Код тіла
return stream;
};
```

Контрольні завдання та запитання

1. Які основні положення системи уведення-виведення C++?
2. Що таке потік?
3. Які потоки відкриваються під час запуску програми мовою C++? Яке їх призначення?
4. Які прапори формату ви знаєте? Яке їх призначення? Наведіть приклади.
5. Які функції, що визначають параметри формату ви знаєте? 6. Яке їх призначення? Наведіть приклади застосування.
7. Які особливості файлового введення-виведення в C++?
8. Які функції виконують оператори вставки та вилучення?
9. Як і для чого створюються функції вставки та вилучення?
10. Яка загальна форма функції вставки?
11. Яка загальна форма функції вилучення?
12. Чи можуть функції вставки та вилучення належати класу, для якого їх створюють?
13. Які типи параметрів та який тип значення, що повертається, у функції вставки та вилучення?
14. Яка мета створення власних маніпуляторів? Наведіть їх загальну форму.

Список використаних джерел

1. Глинський Я. М. C++ і C++ Builder: навч. посібн. / Я. М. Глинський, В. Є. Анохін, В. А. Рижська – Львів: Деол СПД Глинський, 2011. – 192 с.

2. Трофименко О.Г. С++. Основи програмування. Теорія та практика : підручник / О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін. ; за ред. О. Г. Трофименко. – Одеса: Фенікс, 2010. – 544 с.

3. Грицюк Ю.І. Об'єктно-орієнтоване програмування мовою С++: навч. посібн. / Ю. І. Грицюк, Т. Є. Рак — Львів: ЛДУ БЖД, 2011. — 404 с.

4. Страуструп Б. Программирование: принципы и практика использования С++: пер.с англ. – М.: Вильямс, 2012. – 1248 с.

5. Шилдт Г. Справочник программиста по С/С++: пер.с англ. – М.: Вильямс, 2006. 800 с.

6. Павловская Т.А. С/С++. Структурное и объектно-ориентированное программирование: практикум / Т. А. Павловская, Ю. А. Щупак. – СПб.: Питер, 2010. – 352 с.

7. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. / Г. Буч, Р. Максимчук, М. Энгл, Б. Янг, Д. Коннален, К. Хьюстон. – М.: Вильямс, 2018. – 720 с.