# Об’єктно-орієнтованого програмування: основні поняття.

Об'єктно-орієнтоване програмування – це методологія побудови програм, що використовує об'єктну декомпозицію задачі, при якій структура системи описується в термінах об'єктів і зв'язків між ними, а поведінка - в термінах обміну повідомленнями між цими об'єктами.

Головною ідеєю ООП є логічне об’єднання в межах об'єкта даних та операцій над ними.

Фундаментальними в ООП є поняття об'єкта та класу.

Об'єкт - це сукупність атрибутів (інформаційних полів) і методів їхньої обробки (функцій зміни властивостей об'єкта).

Сукупність об'єктів, які мають загальний набір атрибутів (загальну структуру) і однакову поведінку, називають класом.

# Об’єктно-орієнтованого програмування: основні принципи (властивості).

Основні принципи ООП:

Інкапсуляція

Наслідування

Поліморфізм

Інкапсуляція – це принцип приховування даних, згідно з яким будь-який клас розглядається як «чорний ящик», що об’єднує (інкапсулює) в єдиній інформаційній структурі дані і методи їхньої обробки. Користувач класу може бачити і використовувати тільки інтерфейсну частину класу (список задекларованих властивостей і методів класу), що захищає дані від зовнішнього втручання і некоректного використання.

Під спадкуванням розуміють механізм, згідно з яким будь-який клас можна породити від іншого зі збереженням усіх атрибутів і методів класу-предка та додаючи, при необхідності, нові атрибути і методи.

Можливість класу-нащадка змінювати реалізацію методів батьківського класу називають поліморфністю класу, а принцип, згідно з яким родинні об'єкти (об'єкти, що мають одного загального батька) можуть виконувати однотипні дії по-різному, залежно від того, об'єкт якого класу використовується при виклику відповідного методу, - поліморфізмом об'єктів.

# Концепція об’єктно-орієнтованого підходу: складові об’єктно-орієнтованої технології, їх загальна характеристика.

Об'єктно-орієнтований підхід до програмування орієнтований на колективне розроблення великих програмних систем групою програмістів і включає в себе 3 основні компоненти:

об'єктно-орієнтований аналіз (object-oriented analysis, ООА),

об'єктно-орієнтоване проектування (object-oriented design, ООD),

об'єктно-орієнтоване програмування (ООП).

Об'єктно-орієнтований аналіз - це методологія, що виявляє концептуальні сутності предметної області для розуміння і пояснення того, як вони взаємодіють між собою. Об'єктно-орієнтований аналіз спрямований на створення моделей реальної дійсності на основі об'єктно-орієнтованого світогляду.

Об'єктно-орієнтоване проектування - це методологія проектування, що поєднує в собі об'єктну декомпозицію та прийоми представлення логічної (класи і об'єкти) і фізичної (модулі і процеси) моделі (структури) системи, а також її статичні та динамічні аспекти.

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.

# Основи об’єктно-орієнтованих технологій: об’єктна модель, загальна характеристика.

Логічне проектування полягає в розробці структури класів: визначаються атрибути для зберігання складових стану об'єктів і алгоритми методів, що реалізують аспекти поведінки об'єктів. При цьому використовуються розглянуті вище прийоми розробки класів: успадкування, композиція, поліморфізм і т.д. Результатом є ієрархія (діаграма) класів, що відображає взаємозв'язок класів, а також опис класів.

Фізичне проектування включає об'єднання описів класів в модулі, вибір схеми їх підключення (статична або динамічна компоновка), визначення способів взаємодії з обладнанням, з операційною системою і/або іншим програмним забезпеченням (наприклад, базами даних, мережевими програмами), забезпечення синхронізації процесів для систем паралельної обробки і т.ін.

Результатом об'єктно-орієнтованого проектування є побудова діаграми класів (design class diagram) і діаграм взаємодій (collaboration diagram).

# Основи об’єктно-орієнтованих технологій: об’єктна модель, складові об’єктного підходу.

Об'єктно-орієнтований підхід до програмування орієнтований на колективне розроблення великих програмних систем групою програмістів і включає в себе 3 основні компоненти:

об'єктно-орієнтований аналіз (object-oriented analysis, ООА),

об'єктно-орієнтоване проектування (object-oriented design, ООD),

об'єктно-орієнтоване програмування (ООП).

Логічне проектування полягає в розробці структури класів: визначаються атрибути для зберігання складових стану об'єктів і алгоритми методів, що реалізують аспекти поведінки об'єктів. При цьому використовуються розглянуті вище прийоми розробки класів: успадкування, композиція, поліморфізм і т.д. Результатом є ієрархія (діаграма) класів, що відображає взаємозв'язок класів, а також опис класів.

Фізичне проектування включає об'єднання описів класів в модулі, вибір схеми їх підключення (статична або динамічна компоновка), визначення способів взаємодії з обладнанням, з операційною системою і/або іншим програмним забезпеченням (наприклад, базами даних, мережевими програмами), забезпечення синхронізації процесів для систем паралельної обробки і т.ін.

Результатом об'єктно-орієнтованого проектування є побудова діаграми класів (design class diagram) і діаграм взаємодій (collaboration diagram).

# Оголошення класу: формат, специфікатори доступу, їх загальна характеристика.

Оголошення класу являє собою узагальнення комбінованого типу даних і складається з двох частин: заголовка, що включає ключове слово class, і тіла, укладеного у фігурні дужки:

class ім'я\_класу

{ тіло\_класу;

};

Однак, для підтримки принципа інкапсуляції, доступ до членів класу (атрибутів і методів) з об'єктів інших класів може бути обмеженим. Це забезпечуються введенням в клас областей доступу:

- public (відкритий/загальний доступ) – доступ з поза класу не обмежений;

- protected (захищений доступ) – доступ можливий тільки власним методам і методам похідних класів;

- private (закритий/приватний доступ) – доступ можливий тільки власним методам.

Специфікатори доступу – це спеціальні синтаксичні конструкції, що явно задають область видимості кожного члена класу.

# Реалізація класу: зовнішні та вбудовані методи, особливості їх задання.

. Реалізація класу відображає внутрішній його устрій і складається з визначення методів, оголошених в інтерфейсі класу. Кожен метод класу повинен бути визначений (описаний) в програмі. Визначити метод можна або безпосередньо усередині класу (в тілі класу), або поза його межами (після тіла класу).

***Особливості задання зовнішніх методів класу*:**

Оскільки різні класи можуть мати методи з однаковими іменами, при визначенні зовнішнього методу необхідно вказати до якого класу він належить. Для цього треба кваліфікувати ім'я метода іменем відповідного класу за допомогою операції розширення області видимості "::":

тип\_результату ім'я\_класу :: ім'я\_методу ([параметри])

{ тіло };

Оператор "::" «прив'язує» ім'я члена до імені класу, тим самим однозначно ідентифікуючи методи даного класу. Це дозволяє компілятору самостійно визначити до якого класу належить та чи інша функція.

Наприклад, ось як можна записати код зовнішнього методу Set\_Сolor():

void TPoint :: Set\_Сolor(int color\_new)

{ color = color\_new;

}

***Особливості задання вбудованих методів класу*:**

Будь-яка функція, яка визначається в оголошенні класу, автоматично стає вбудо-ваною. У цьому випадку необов'язково специфікувати її оголошення ключовим сло-вом inline.

class TPoint

{ int x, y;

int color;

public :

int GetX() { cout<<" X= "<< x;} // автоматично вбудована функціїя

int GetY() { cout<<" Y= "<< y;}

int Get\_Сolor() { cout<<" Сolor - "<< color;}

void Set\_XY(int хх, int yy) { х = хх; y = yy; }

void Set\_Сolor(int c) { color = c; }

};

# Організація доступу до членів об'єкта: способи створення об'єктів, особливості реалізації доступу до їх членів.

Створення об'єкта в програмі здійснюється оголошенням відповідної змінної (для статичних об'єктів) або динамічним розміщенням змінної в пам'яті (для динамічних об'єктів). Способи створення таких об'єктів залежать від наявності або відсутності в класі спеціального методу, що визначає необхідні для створення об'єкта дії, – так званого конструктора.

1. Якщо в класі відсутній конструктор, але описані захищені (protected) або прихо-вані (private) атрибути, то можливе створення тільки неініціалізованих об'єктів.

Формат створення об'єктів:

ім'я\_класу ім'я\_об'єкта;

ім'я\_класу \* покажчик\_на\_об'єкт = new ім'я\_класу;

1. За відсутності в класі конструктора і захищених (protected) або прихованих (pri-vate) полів для оголошення ініціалізованих об'єктів використовують оператор ініці-алізації, застосовуваний при створенні ініціалізованих структур, наприклад,

TPoint рoint = {3, 4, 2};

Ініціалізуючі значення при цьому повинні перераховуватися в порядку задання полів в описі класу.

1. **За допомогою конструктора**

Реалізація доступу до членів класу визначається областю їх видимості відносно коду (методу), з якого здійснюється звернення.

Звертання до відкритих (public) членів класу (як атрибутів, так і методів) ззовні об'єкта даного класу (з методів інших класів або з функцій, не пов'язаних ні з яким класом), здійснюється за допомогою повних імен, кожне з яких має вигляд:

ім'я\_об'єкта . ім'я\_класу :: ім'я\_члена\_класу;

Якщо атрибут оголошений як private або protected , то доступ до нього із методів того ж самого класу здійснюється аналогічно, як і до відкритих (public) атрибутів цього класу (за іменем, без явного вказання імені об'єкта).

Доступ же до таких атрибутів з методів інших класів або з функцій, не пов'язаних ні з яким класом, здійснюється тільки через методи цього ж класу.

# Структури як різновиди класів: особливості визначення, порівняльна характеристика.

Синтаксис класу в С++ збігається з синтаксисом структури С++:

struct ім'я\_структури

{ визначення\_членів\_структури;

}

По суті, класи і структури є спорідненими типами. За одним винятком, вони взає-мозамінні, оскільки структура також може містити дані та програмні коди, які мані-пулюють цими даними так само, як це може робити і клас. Єдина відмінність між С++-структурою і С++-класом полягає у тому, що за замовчуванням члени класу є закритими (private), а члени структури – відкритими (public).

# Об'єднання як різновиди класів: особливості визначення, порівняльна характеристика.

Згідно з визначенням мови програмування C++, об'єднання – це, по суті, аналогічний клас, у якому всі члени даних зберігаються в одній і тій самій області. Об'єднання може містити конструктор і деструктор, а також функції-члени. Звичайно ж, члени об'єднання за замовчуванням відкриті (public), а не закриті (private).

# Конструктори: поняття конструктора, класифікація конструкторів.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

Загалом конструктори можна класифікувати наступними чином:

1) за призначенням

- ініціалізуючий конструктор,

- неініціалізуючий конструктор,

2) за наявністю параметрів

- з параметрами,

- без параметрів,

3) за кількістю та типом параметрів

- конструктор за замовчуванням,

- конструктор з двома і більше параметрами,

- конструктор копіювання,

- конструктор перетворення.

# Конструктори: поняття конструктора, конструктор за замовчуванням, особливості реалізації і використання.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

Конструктор, який не має параметрів, називають конструктором за замовчуванням. Такий конструктор, зазвичай, ініціалізує атрибути класу константними значеннями.

Формат визначення конструктора за замовчуванням:

ім'я\_класу()

{ тіло\_конструктора

}

Якщо в класі явно не описано жодного конструктора, то компілятор автоматично генерує «порожній» конструктор, який теж грає роль конструктора за замовчуванням.

Клас може містити тільки один конструктор, що дозволяє створити об'єкт без вказання аргументів. Якщо б таких конструкторів було декілька, то компілятор не зміг би вирішити, який з них повинен бути викликаний.

Особливо важлива наявність такого конструктора при створенні масиву об'єктів.

# Конструктори: поняття конструктора, конструктор з двома і більше параметрами, особливості реалізації і використання.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

Конструктор, який разом із створенням об'єкту присвоює його атрибутам деякі початкові значення, отримуючи їх як параметри, називають конструктором з параметрами.

Формат визначення конструктора з двома і більше параметрами:

ім'я\_класу(тип параметр\_1, …, тип параметр\_n)

{ тіло\_конструктора

}

# Конструктори: поняття конструктора, конструктор копіювання, особливості реалізації і використання.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

При створенні об'єкта його атрибути можуть бути проініціалізовані значеннями атрибутів іншого об'єкта цього ж типу. Для такого створення об'єкту у С++ викорис-товується так званий конструктор копіювання.

Формат визначення конструктора копіювання:

ім'я\_класу\_1([const] ім'я\_класу\_1 &ім'я\_об'єкта);

Якщо у класі немає явно описаного конструктора копіювання, то компілятор генерує його автоматично. Такий конструктор виконує *поверхневе копіювання*, тобто поелементне копіювання атрибутів класу.

# Конструктори: поняття конструктора, конструктор перетворення, особливості реалізації і використання.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

При створенні об'єкта його можна проініціалізувати об'єктом іншого типу, тобто перетворити об’єкт, тип якого відрізняється від типу конструктора, в об’єкт даного класу. Для такого створення об'єкту використовується конструктор перетворення.

Можливі формати заголовка конструктора перетворення:

ім'я\_класу(ім'я\_класу ім'я\_об'єкта);

ім'я\_класу(ім'я\_класу &ім'я\_об'єкта);

ім'я\_класу(const ім'я\_класу &ім'я\_об'єкта);

Конструктор перетворення може викликатися автоматично. Таке перетворення типу неявно застосовується тільки якщо воно унікальне.

# Конструктори: поняття конструктора, список ініціалізаціі, особливості його задання і використання.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

Для таких випадків застосовується спеціальна конструкція, що включається в опис конструктора, - список ініціалізації, який дозволяє ініціалізувати змінні без ви-користання операцій присвоєння значень. Формат конструктора зі списком ініціалі-зації:

ім'я\_класу(параметри) : список\_ініціалізаціі

{ тіло\_конструктора

}

Конструктор з параметрами, що містить список ініціалізації часто називаюють конструктором ініціалізації.

Основне призначення даного конструктора – ініціалізація атрибутів особливих типів, наприклад, константних і посилальних.

# Конструктори: поняття конструктора, перевантаження конструкторів.

Конструктор класу – це спеціальний відкритий (public) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

У класі може бути визначено декілька конструкторів з одним і тим самим іменем. Тоді конструктор вважається перевантаженим. Єдина вимога: не може бути двох конструкторів з однаковим (за типами) набором формальних параметрів, зокрема, не може бути двох конструкторів за умовчанням.

# Конструктори: особливості використання.

конструктор имеет тоже имя, что и его класс;

– конструктор не содержит оператора return и не возвращает никакого значения (даже void);

– конструктор может определяться явно пользователем или неявно компилятором. Неявно происходит объявление конструктора в таких случаях: определение нового объекта класса, при копировании объекта, при динамическом создании объекта с помощью оператора new

* определение конструктора может производиться внутри и вне класса;

– конструктор может иметь, а может не иметь параметры. Конструктор с параметрами инициализирует объект при объявлении. Конструктор без параметров инициализирует пустой объект;

– конструкторы не могут быть вложенными;

– конструктор не может быть константным, статическим или виртуальным (const, static или virtual);

– ошибки, генерируемые конструктором, обрабатываются только механизмом обработки исключительных ситуаций;

– конструктор не наследуется.

# Деструктори: поняття деструктора, особливості реалізації.

Деструктор – спеціальний метод, який автоматично викликається для коректного знищення об'єктів.

Ім'я деструктора, як і ім'я конструктора, співпадає з іменем классу, але йому пере-дує ще додатковий символ "~" («тильда»):

~ім'я\_класу() { … }.

У деструкторів немає параметрів. Подібно до конструкторів, деструктори не по-вертають значень, а отже, в їх оголошеннях відсутній тип значення, що поверта-ється.

Клас може мати тільки один деструктор або не мати жодного.

# Динамічні об'єкти: класи з динамічними атрибутами, особливості створення, ініціалізація і знищення динамічних об'єктів.

При створенні класів з динамічними полями необхідно враховувати велику кількість особливостей:

* Наявність свого конструктора
* Деструктора
* Конструктора копіювання
* Перевантаженого оператора «=»

При виділенні динамічної пам'яті для окремих об'єктів використовують такі форми звернення до операції new:

ім'я\_покажчика\_на\_об'єкт = new ім'я\_класу;

ім'я\_покажчика\_на\_об'єкт = new ім'я\_класу (список\_параметрів);

Другу форму застосовують за наявності у конструктора списку параметрів.

Операція delete вимагає вказання тільки імені об'єкта:

delete ім'я\_покажчика\_на\_об'єкт;

# Особливості роботи з об'єктами: присвоєння об'єктів, способи реалізації.

Об'єкти одного і того ж класу можна присвоювати один одному. При виконанні операції присвоювання за замовчуванням здійснюється поверхневе копіювання, тобто дані першого об'єкта порозрядно копіюються у другий. Тому якщо об’єкт містить динамічні атрибути, то необхідно перевантажити конструктор копіювання та оператор «=».

# Особливості роботи з об'єктами: способи передачі об'єктів функціям, їх загальна характеристика.

Об'єкти можна передавати у функції так само, як і змінні будь-яких інших типів даних. За замовчуванням об'єкти класу передаються функціям за значенням. Тобто, у функцію передається не сам об'єкт, а його копія, яка стає параметром функції. Створення копії означає появу нового тимчасового об'єкта. Усі зміни, внесені в об'єкт-копію в процесі виконання функції, не впливають на вхідний об'єкт, який використовується як аргумент для функції.

Якщо в класі безпосередньо не визначено конструктор копіювання, то він генерується за замовчуванням. Конструктор копіювання за замовчуванням створює побітову (тобто однакову) копію об'єкта.

Коли виконання функції, яка містить аргумент-об'єкт завершується, копія аргумента руйнується, для чого викликається деструктор цього локального об'єкта. Потреба виклику даного деструктора пов'язана з виходом локального тимчасового об'єкта з області видимості функції, у якій він використовувався.

Також можна передавати об’єкти у функції за посиланнями та покажчиками. Тоді їх побітові копії не створюються.

# Особливості роботи з об'єктами: способи повернення об'єктів функціями, їх загальна характеристика.

Щодо механізму повернення об'єктів функціями, то тут виникає ситуація, анало-гічна ситуації з передачею об'єктів у функції. Якщо тип значення, що повертається функцією, є об’єктом класу, то під час виклику функції компілятор автоматично ге-нерує тимчасовий об'єкт цього класу і використовує значення, яке визначено в опе-раторі return, для ініціалізації цього об’єкта. Після повернення значення із функції негайно викликається деструктор тимчасового об'єкта і цей об'єкт руйнується.

Руйнування тимчасового об'єкта в деяких ситуаціях може викликати непередба-чувані побічні ефекти. Так, якщо повернутий функцією об'єкт має деструктор, який звільняє динамічно виділену область пам'яті, то ця пам'ять буде звільнена навіть у тому випадку, якщо об'єкт, який отримує повернуте функцією значення, все ще ви-користовує цю пам'ять.

Одним із варіантів вирішення цієї проблеми є використання для повернення покажчика або посилання на об'єкт. Але здійснити це не завжди вдається.

Ще один спосіб вирішення цієї проблеми полягає у використанні конструктора копіювання.

# Особливості роботи з об'єктами: неявний параметр *this*.

Коли метод, що належить класу, викликається для обробки даних конкретного об'єкта, йому, крім явно оголошених параметрів, автоматично і неявно передається ще один прихований параметр: покажчик на той об'єкт, для якого викликається цей метод. У С++ цей покажчик має спеціальне ім'я this і неявно визначений в кожному методі класу наступним чином:

ім'я\_класу \*const this = адреса оброблюваного об'єкта.

Явно описати чи визначити вказівник this не можна і не потрібно. Відповідно до неявного визначення, this є константним покажчиком, тобто змінювати його не можна.

# Особливості роботи з об'єктами: константні члени класу і константні об'єкти, особливості використання.

Коли метод, що належить класу, викликається для обробки даних конкретного об'є-кта, йому, крім явно оголошених параметрів, автоматично і неявно передається ще один прихований параметр: покажчик на той об'єкт, для якого викликається цей ме-тод. У С++ цей покажчик має спеціальне ім'я this і неявно визначений в кожному методі класу наступним чином:

ім'я\_класу \*const this = адреса оброблюваного об'єкта.

# Особливості роботи з об'єктами: статичні члени класу, особливості використання.

Члени класу (як методи, так і атрибути) можуть бути загальними (спільними) для усіх об'єктів даного класу. Тоді їх називають статичними.

Статичні атрибути використовуються для збереження даних, спільних для всіх ек-земплярів даного класу (значення яких є однаковими для всіх об'єктів даного класу).

Створення статичних атрибутів виконується в три етапи:

- оголошення (декларація);

- визначення (виділення пам’яті);

- ініціалізація.

Особливістю визначення статичного атрибута є те, що повторювати ключове слово static при цьому не можна. Воно повинно бути присутнім тільки в тілі класу.

Ініціалізацію статичних атрибутів можна поєднати з їх визначенням. Формат такого визначення:

тип\_змінної ім'я\_класу :: ідентифікатор [= ініціалізатор];

Cтатичний атрибут може належати до типу того ж класу, членом якого він є.

Також статичний атрибут може виступати в ролі аргументу за замовчуванням для метода класу, а для нестатичних це заборонено.

Статичні методи – це методи, які є спільними для усіх об'єктів даного класу.

Статичні методи класу вирізняються тим, що мають доступ тільки до статичних членів (атрибутів і методів) даного класу.

# Особливості роботи з об'єктами: статичні атрибути, особливості створення і використання.

Члени класу (як методи, так і атрибути) можуть бути загальними (спільними) для усіх об'єктів даного класу. Тоді їх називають статичними.

Статичні атрибути використовуються для збереження даних, спільних для всіх ек-земплярів даного класу (значення яких є однаковими для всіх об'єктів даного класу).

Створення статичних атрибутів виконується в три етапи:

- оголошення (декларація);

- визначення (виділення пам’яті);

- ініціалізація.

Особливістю визначення статичного атрибута є те, що повторювати ключове слово static при цьому не можна. Воно повинно бути присутнім тільки в тілі класу.

Ініціалізацію статичних атрибутів можна поєднати з їх визначенням. Формат такого визначення:

тип\_змінної ім'я\_класу :: ідентифікатор [= ініціалізатор];

Cтатичний атрибут може належати до типу того ж класу, членом якого він є.

Також статичний атрибут може виступати в ролі аргументу за замовчуванням для метода класу, а для нестатичних це заборонено.

# Особливості роботи з об'єктами: статичні методи, особливості створення і використання.

Статичні методи – це методи, які є спільними для усіх об'єктів даного класу.

Статичні методи класу вирізняються тим, що мають доступ тільки до статичних членів (атрибутів і методів) даного класу.

# Дружні функції і дружні класи, особливості реалізації.

Дружньою функцією класу називають функцію, яка, не будучи членом класу, має доступ до усіх його членів, у тому числі і захищених.

Для того щоб оголосити зовнішню функцію дружньою даному класу, то необхідно помістити її прототип з ключовим словом friend в тіло класу.

Працюючи з дружніми функціями, потрібно враховувати наступні особливості:

- для дружньої функції несуттєві специфікації доступу;

- дружня функція при виклику не отримує покажчика this;

- об'єкти класів повинні передаватися дружній функції тільки явно через апарат параметрів;

- дружні функції не можуть мати специфікатори static і extern;

- у класі, функція-член якого є дружньою до не цілком оголошеного класу, слід розміщати тільки прототип; реалізація функції повинна знаходитися після пов-ного оголошення класу;

- при виклику дружньої функції не можна використовувати операції вибору:

імя\_об'екта.імя\_функціі

указатель\_на\_об'ект-> імя\_функціі

- дружні функції не взаємні;

- одна і та ж функція може бути "другом" декількох класів;

- дружні функції не успадковуються.

Коли усі функції деякого класу є дружніми стосовно іншого класу, можна оголосити весь клас ***дружнім***.

# Масиви об'єктів: особливості створення і ініціалізації.

Якщо при створенні масиву використовувався неініціалізуючий конструктор, то всі об'єкти - елементи масиву будуть неініціалізованими. Застосування ініціалізуючого конструктора без параметрів або конструктора з параметрами, заданими за умовчанням, може:

- ініціалізувати всі створювані об'єкти однаково, як задано в ініціалізуючому конструкторі (рідко потрібно за умовою задачі);

- ініціалізувати всі створювані об'єкти випадковим чином (ще рідше потрібно за умовою задачі);

- ініціалізувати всі створювані об'єкти значеннями, що вводяться в процесі ініціалізації (введення в методі ініціалізації вважається нетехнологічним).

Приклад:

myClass array[4][2] = { myClass(1, 2), myClass(3, 4),

myClass(5, 6), myClass(7, 8),

myClass(9, 10), myClass(11, 12),

myClass(13, 14), myClass(15, 16) };

# Динамічні масиви об'єктів: способи створення і знищення, їх загальна характеристика.

Масиви можуть формуватися і з динамічних об'єктів. Це можна зробити трьома способами:

- створити динамічний масив об'єктів - пам'ять під нього виділяють одним безпе-рервним фрагментом рівним обсягу всіх об'єктів масиву, наприклад,

B mas[] = new B[n];

- створити статичний масив покажчиків на об'єкти і потім динамічно виділити пам'ять під елементи, наприклад,

B \*mas[n]; // пам'ять під масив покажчиків виділена статично

for (i=0; i<n; i++) mas[i] = new B; // виділення пам'яті під об'єкти

- створити динамічний масив покажчиків і потім також динамічно виділити па-м'ять під елементи наприклад,

B \*\*mas = new B \*[n]; // пам'ять під масив покажчиків виділена динамічно

for (i=0; i<n; i++) mas[i] = new B; // виділення пам'яті під об'єкти

Звільняти виділену пам'ять потрібно так, як вона була виділена:

- одним фрагментом - для динамічного масиву об'єктів:

delete [] mas;

- поелементно - для масиву покажчиків на об'єкти:

for (i = 0; i <n; i ++) delete mas [i];

- одним фрагментом, якщо пам'ять під масив покажчиків виділялася динамічно:

delete [] mas;

# Реалізація успадкування в классах: різновиди наслідування, їх загальна характеристика.

В основі механізму, що дозволяє створювати ієрархії класів, лежить принцип успадкування - здатність одного класу успадковувати характеристики іншого класу.

Механізм успадкуванням передбачає конструювання нових похідних класів (класів-нащадків) з вже наявних базових класів (класів-батьків) шляхом автоматичного успадкування членів базового класу і додавання власних атрибутів і методів.

У функціональному сенсі похідні класи є більш потужними по відношенню до базових класів, оскільки, включаючи атрибути і методи базового класу, вони мають ще й свої компоненти.

Успадкування буває одиночним і множинним. При одиночному успадкуванні в кожного похідного класу є лише один базовий клас, а при множинному — декілька.

# Просте успадкування, його різновиди.

Для визначення похідного класу використовується наступна синтаксична конструкція:

class похідний\_клас : [специфікатор\_доступу] базовий\_клас

{ тіло\_нового\_класу

}

У такому оголошенні похідного класу елемент специфікатор\_доступу є необов'язковим. У разі ж потреби він може бути визначається одним із специфікато-рів: public, private або protected.

Керуючись оголошенням похідного класу, компілятор ніби збирає його з різних частин — спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного класу

# Різновиди простого успадкування: відкрите успадкування, особливості реалізації.

| **Специфікатор доступу до елементів у базовому класі** | **Тип успадкування** | | |
| --- | --- | --- | --- |
| public | protected | private |
| *public* | public | protected | private |
| *protected* | protected | protected | private |
| *private* | не доступні | не доступні | не доступні |

Як видно із прикладу, відкрите успадкування дозволяє обійтися без створення об'єкта класу baseClass. Звернутися до методів цього базового класу можна безпосередньо через об'єкт похідного класу derived:

derived ObjD(3);

ObjD.setB(1, 2);

ObjD.showB("Base class: ");

# Різновиди простого успадкування: закрите успадкування, особливості реалізації.

| **Специфікатор доступу до елементів у базовому класі** | **Тип успадкування** | | |
| --- | --- | --- | --- |
| public | protected | private |
| *public* | public | protected | private |
| *protected* | protected | protected | private |
| *private* | не доступні | не доступні | не доступні |

Наведений вище код програми не відкомпілюється, оскільки функції setB() і showB() тепер стали private-членами класу derived (тобто, доступними тільки для методів похідного класу), і тому їх не можна викликати з функції main().

# Різновиди простого успадкування: захищене успадкування, особливості реалізації.

| **Специфікатор доступу до елементів у базовому класі** | **Тип успадкування** | | |
| --- | --- | --- | --- |
| public | protected | private |
| *public* | public | protected | private |
| *protected* | protected | protected | private |
| *private* | не доступні | не доступні | не доступні |

Якщо базовий клас успадковується як захищений (protected), то всі його відкриті і захищені члени стають захищеними членами похідного класу. Вони видимі лише функціям базового і похідного класів і є невидимими з інших точок програми.

У цьому випадку члени (класу baseClass) setB() і showB() стають protected-членами класу derived. Це означає, що до них не можна отримати доступу з коду програми, "прописаного" поза класом derived. Тому посилання на ці члени у функції main() (через об'єкт ObjD) неправомірні.

# Механізми використання захищених членів класу, особливості реалізації.

При успадкуванні ж класів механізм успадкування захищеного члена класу істотно відрізняється від механізму успадкування закритого члена класу.

Як відомо, закриті члени базового класу не доступні ніяким іншим частинам програми, окрім методів свого класу, в т.ч. йдеться і про похідні класи. Із захищеними членами класу все відбувається інакше. Зокрема, якщо базовий клас успадковується як public-клас, то захищені члени базового класу стають захищеними членами похідного класу, тобто доступними для нього.

Отже, використовуючи специфікатор доступу protected, можна створити члени класу, які закриті у межах свого класу, але можуть успадковуватися похідним класом з отриманням доступу до себе.

# Просте успадкування: повернення успадкованим членам класу початкової специфікації доступу, основні способи, особливості їх реалізації.

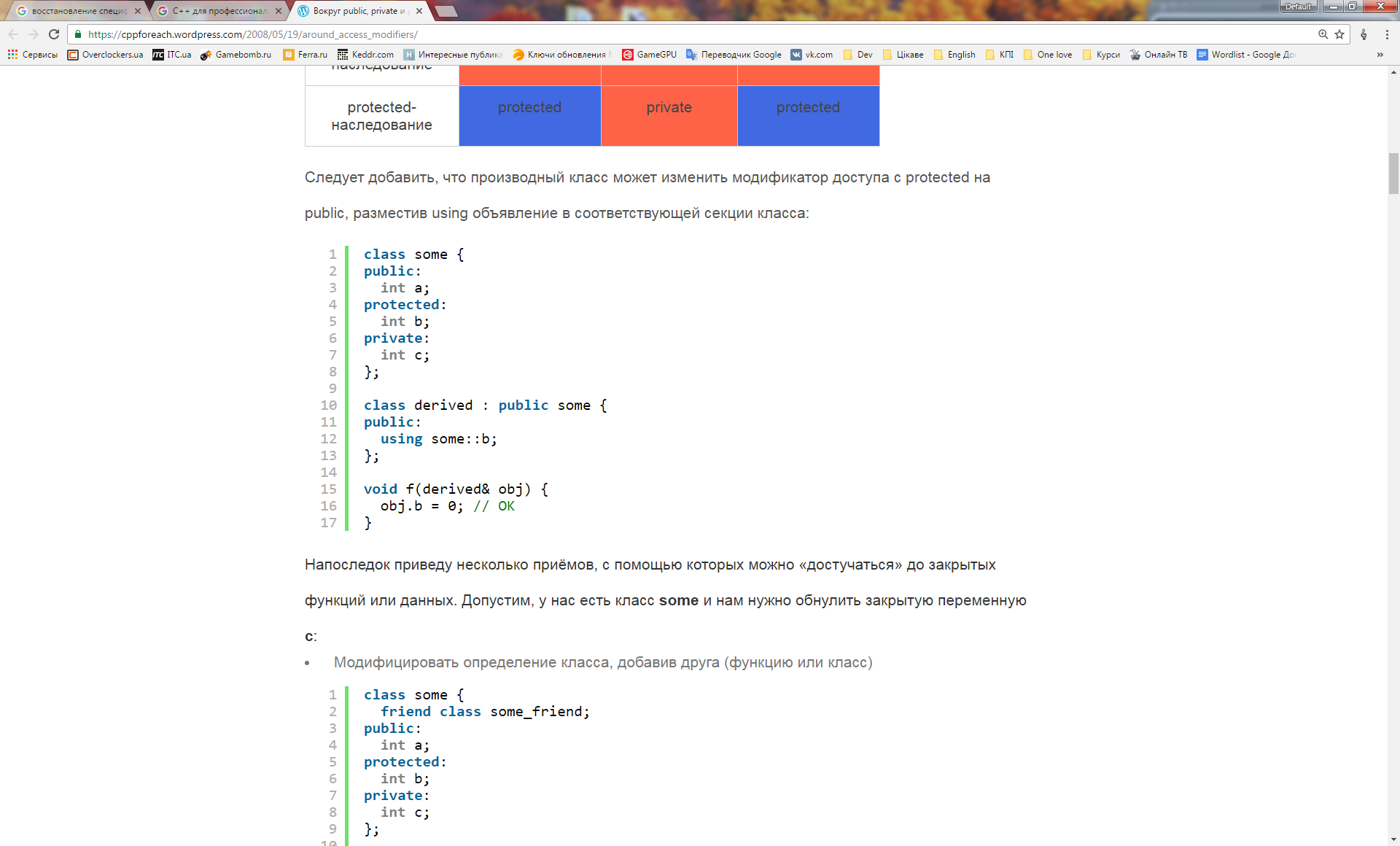
Коли базовий клас успадковується як закритий (як private-клас), то всі його члени (відкриті, захищені та закриті) стають private-членами похідного класу. Але за певних обставин один або декілька успадкованих членів необхідно повернути до їх початкової специфікації доступу. Наприклад, незважаючи на те, що базовий клас успадковується як private-клас, деяким визначеним його public-членам потрібно надати public-статус у похідному класі. Це можна зробити двома способами:

* по-перше, у похідному класі можна використовувати оголошення using;
* по-друге, можна налагодити доступ до успадкованого члена за допомогою *оголошень доступу.*

Оголошення доступу має такий формат:

базовий\_клас::член;

Оголошення доступу можна використовувати для відновлення прав доступу public- і protected-членів. Проте для зміни (підвищення або пониження) статусу доступу його використовувати не можна. Наприклад, член, оголошений закритим у базовому класі, не можна зробити відкритим у похідному.



# Просте успадкування: використання конструкторів і деструкторів.

Конструктори і деструктори базового класу в похідних класах не успадковуються. Однак, якщо базовий клас містить хоча б один конструктор і деструктор, то похідний клас також повинен включати власні конструктор і деструктор. При цьому С++ підтримує певні правила взаємодії між цими компонентами базових і похідних класів.

Так, при створенні об'єктів похідного класу передбачений автоматичний виклик конструктора базового класу для ініціалізації його полів (явний виклик конструктора базового класу в програмі неможливий). Це і зрозуміло — об'єкт похідного класу являє собою модифікований об'єкт базового класу. Він не може виникнути з нічого. Отже, спочатку викликається конструктор базового класу, а потім — конструктор похідного класу.

Таким чином, при створенні об'єкта похідного класу спочатку викликається конструктор базового класу, а за ним – конструктор похідного класу. Під час руйнування об'єкта похідного класу спочатку викликається його "рідний" конструктор, а за ним – конструктор базового класу.

# Механізми множинного успадкування, особливості реалізації, порядок виклику конструкторів і деструкторів.

Похідний клас може успадковувати не тільки один, а й одночасно декілька класів. Такий спосіб породження класу називають множинним успадкуванням.

Опис похідного класу при множинному успадкуванні виглядає наступним чином:

class похідний\_клас: специфікатор\_доступу базовий\_клас\_1,

...,

специфікатор\_доступу базовий\_клас\_N

{ тіло\_класу

};

Як видно, щоб забезпечити успадкування декількох базових класів, необхідно через кому перерахувати їх імена у вигляді списку. Вид спадкування, як і у випадку простого спадкування, визначається режимом доступу до компонентів відповідного базового класу.

Послідовність активізації конструкторів при множинному успадкуванні (коли похідний клас успадковує декілька базових класів) така ж, як і для випадку єдиного базового класу: спочатку активізуються конструктори всіх базових класів в порядку їх перерахування в оголошенні похідного класу, потім конструктори об'єктних полів і на завершення - конструктор похідного класу

# Віртуальне наслідування, особливості реалізації.

Множинне успадкування часто приводить до неоднозначностей. Розглянемо класичний варіант, що одержав назву діамантового успадкування, — клас baseClass є базовим для класів derivedA і derivedB, а ті, у свою чергу, — базовими для класу derivedC. Відповідно до механізму успадкування всі члени класу baseClass стають членами класів derivedA і derivedB, потім вони успадковуються класом derivedC. Оскільки при першому успадкуванні (одиночному) члени класу baseClass ”роздвоюються”, при множинному успадкуванні в класі derivedC виникає неоднозначність — як бути з атрибутами класу baseClass.

Уникнути багаторазового включення в похідний клас компонентів базового класу можна двома способами.

Перший полягає у застосуванні оператора дозволу контексту (дозволу області видимості), за допомогою якого можна "вручну" вказати потрібний член.

Більш елегантний спосіб запобігти дублюванню компонентів базового класу в похідному класі - це використання так званого віртуального успадкування.

При віртуальному успадкуванні похідний клас описують наступним чином:

class похідний класс : virtual специфікатор\_доступу базовий\_клас

{ ... };

У цьому випадку включення в похідний клас атрибутів базового класу здійснюється один раз, а їх ініціалізація виконується в похідному класі, який не є прямим нащадком базового класу. Виклик конструкторів при віртуальному успадкуванні відбувається в наступному порядку: спочатку викликається конструктор віртуально успадкованого базового класу, потім конструктори базових класів в порядку їх перерахування при оголошенні похідного класу, за ними - конструктори об'єктних полів і конструктор похідного класу. Деструктори відповідно викликаються в зворотному порядку.

Віртуально наслідуваний клас обов'язково повинен містити конструктор без параметрів, який активізується при виконанні конструкторів класів – прямих нащадків віртуально успадкованого класу.

# Поліморфізм: поняття, різновиди, їх загальна характеристика.

Під поліморфізмом розуміють механізм реалізації методів, при якому методу з одним і тим же ім'ям може відповідати різний програмний код (поліморфний код) в залежності від того, об'єкт якого класу використовується при виклику даного методу.

У різних мовах програмування поліморфізм реалізується різними способами. Зокрема, у С++ підтримується два його різновиди: простий поліморфізм, що базується на механізмі раннього зв'язування, і динамічний поліморфізм, що використовує механізм пізнього зв'язування.

# Статичний поліморфізм, особливості реалізації.

При компіляції програми для коду звичайної функції (не покажчика на функцію) виділяється пам'ять і призначаються адреси для кожного її оператора. Перша адреса у визначенні є адресою функції. При виклику функції, процесор переходить на цю адресу і починає виконувати тіло функції. Найважливішим тут є те, що адреса функції призначається під час компіляції програми, і саме ця адреса використовується в подальшому при виклику функції.

Такий механізм виклику функції називають раннім (або статичним) зв'язуванням. Раннє зв'язування означає, що вся інформація, необхідна для виклику функції, відома при компілюванні програми.

Оскільки при перевантаженні методів реалізується раннє (статичне) зв'язування (вся інформація, необхідна для виклику методів, відома на етапі компіляції програми), то такий різновид поліморфізму називають простим (або статичним) поліморфізмом.

Таким чином, статичий поліморфізм підтримується мовою С++ на етапі компіляції і реалізується за допомогою механізму перевизначення (перевантаження) методів. Тому такі поліморфні методи називаються в С++ перевизначеними. У відповідності із загальними правилами перевизначення функцій такі методи повинні відрізнятися сигнатурою, тобто кількістю, типом і порядком перерахування переданих параметрів.

Перевантаження методів дозволяє будувати гнучкі і досконалі ієрархії класів, перевизначаючи в похідних класах методи відповідно до вимог розроблюваної програми або системи, що використовує ці класи.

Але, використання перевизначених методів не завжди безпечне. Зокрема, відомі випадки, коли при перевизначенні методів виникають помилки, пов'язані з некоректним визначенням типу об'єкта на етапі компіляції, а отже і необхідного аспекта метода, що викликається.

У всіх трьох випадках об'єкт, для якого викликається метод, визначається адресою, що зберігається в покажчику, який за типом може не збігатися з типом об'єкта:

1-й випадок - якщо наслідуваний метод для об'єкта похідного класу викликає метод, перевизначений в похідному класі - в цьому випадку викликаючий метод визначений у базовому класі, тому компілятор вважає, що і аспект поліморфного методу необхідний базового класу, що не коректно для випадку, коли наслідуваний метод викликаний для об'єкта похідного класу.

2-й випадок - якщо об'єкт похідного класу через покажчик базового класу звертається до методу, перевизначеному похідним класом - в цьому випадку тип об'єкта компілятор визначає за типом покажчика, відповідно підключаючи аспект поліморфного методу базового класу, хоча реально об'єкт може належати похідному класу.

3-й випадок - якщо процедура викликає перевизначений метод для об'єкта похідного класу, переданого в процедуру через параметр-змінну, описаний як об'єкт базового класу («процедура з поліморфним об'єктом») - в цьому випадку необхідний аспект поліморфного методу визначається типом параметра змінної, що не коректно, якщо в якості аргументу в процедурі передається об'єкт похідного класу.

# Динамічний поліморфізм, особливості реалізації.

Основою для динамічного поліморфізму слугує покажчик на базовий клас. Покажчики на базові та похідні класи пов'язані такими відносинами, які не властиві покажчикам інших типів. Покажчик одного типу, як правило, не може вказувати на об'єкт іншого типу. Проте покажчики на об'єкти базових класів і об'єкти похідних класів – винятки з цього правила.

У C++ покажчик на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, виведеного з базового. Наприклад, у нас є базовий клас baseClass і похідний клас derivedClass, який виведено з класу baseClass. У C++ будь-який покажчик, оголошений як покажчик на базовий клас baseClass, може бути також покажчиком на похідний клас derivedClass

Якщо виникає потреба за допомогою покажчика на базовий клас отримати доступ до елементів, що визначені певним похідним класом, то необхідно привести цей покажчик до типу покажчика на похідний тип. Наприклад, у процесі виконання цього рядка коду програми дійсно буде викликано функцію showTitle() об'єкта ObjD:

((derivClass \*)bp)->showTitle();

Покажчик можна інкрементувати та декрементувати щодо свого базового типу. Якщо покажчик на базовий клас використовують для доступу до об'єкта похідного типу, то механізм інкрементування або декрементування не примусить його посилатися на наступний об'єкт похідного класу. Натомість він вказуватиме на наступний об'єкт базового класу. Таким чином, інкрементування або декрементування покажчика на базовий клас необхідно розцінювати як некоректну операцію, якщо цей покажчик використовують для посилання на об'єкт похідного класу.

Той факт, що покажчик на базовий тип можна використовувати для посилання на будь-який об'єкт, який є виведеним з базового типу, надзвичайно важливий і принциповий для мови C++. Ця гнучкість є ключовим моментом для механізму реалізації динамічного поліморфізму у C++.

Слід зазначити, що подібно до покажчиків, посилання на базовий клас також можна використовувати для доступу до об'єкта похідного типу. Ця можливість особливо часто застосовується при передачі аргументів функціям. Параметр, який має тип посилання на базовий клас, може приймати об'єкти базового класу, а також об'єкти будь-якого іншого типу, виведеного з нього.

# Механізм віртуальних функцій, особливості реалізації.

Реалізація динамічного поліморфізму базується на використанні двох засобів: механізму успадкування і так званих віртуальних функцій.

Віртуальною називають функцію, яка оголошується в базовому класі з використанням ключового слова virtual і перевизначається в одному або декількох похідних класах. При цьому прототипи віртуальних функцій у різних класах повинні збігатися не тільки за іменами, а й по сигнатурі, хоча алгоритми, реалізовані такими функціями, різні. Якщо прототипи функцій не збігаються, то механізм віртуальності для них не включається.

Формат визначення віртуальної функції:

virtual тип\_результату ім'я\_функції (параметри)

{ тіло\_функції

}

Функція, оголошена віртуальною, залишається віртуальною і у всіх спадкоємцях. Однак іноді в одному або декількох похідних класах перевизначення віртуальної функції може бути відсутнім. У цьому випадку механізм підключення віртуальної функції зберігається, а, отже, викликається функція базового класу, найближчого до розглянутого.

Виклик віртуального метода здійснюється за допомогою так званої таблиці віртуальних методів (Virtual Method Table, VMT) - спеціальної структури в пам'яті, яка автоматично створюється під час компіляції, а потім використовується під час виконання програми. Дана таблиця генерується для кожного класу, що має хоча б один віртуальний метод, і містить адреси віртуальних методів цього класу.

Кожен об'єкт класу, що має віртуальні методи, як прихований атрибут (зазвичай, як перший член), містить покажчик на VMT-таблицю свого класу, який називають вказівником віртуальної таблиці (Virtual Table Pointer, vpointer) або віртуальним табличним курсором. Цей vpointer-вказівник автоматично вставляється у початок конструктора класу на етапі компіляції і додається до об'єкта при його (об'єкта) ініціалізації. Саме при ініціалізації встановлюється взаємозв'язок між об'єктом і його VMT-таблицею.

# Суто віртуальні функції та абстрактні класи, особливості реалізації.

Суто віртуальна функція – це віртуальна функція, яку оголошено в базовому класі, але вона не має у ньому ніякого визначення.

Щоб оголосити суто віртуальну функцію, використовують такий загальний формат:

virtual тип\_результата ім'я\_функції (перелік\_параметрів) = 0;

Клас, у якому є хоча б одна чиста віртуальна функція, називається абстрактним. Призначення абстрактного класу - забезпечити інтерфейс з іншими класами.

# Шаблонні функції: поняття шаблонної функції, виведення її аргументів, спеціалізація та конкретизація шаблонної функції.

Шаблони - це спосіб написання єдиного узагальненого визначення функції або класу, яке компілятор автоматично транслює в спеціальну версію функції або класу для кожного з типів даних, використаних програмою.

У свою чергу, шаблонна функція (template function) визначає універсальну сукупність операцій, які застосовуються до різних типів даних. Типи даних, з якими працює функція, передаються як параметри. Шаблонна функція оголошується за допомогою ключового слова template. Її визначення виглядає так.

template <typename T> тип\_результата ім’я\_функції (список\_параметрів)

{ тіло\_функції

}

Конкретна версія шаблонної функції, створювана компілятором, називається спеціалізацією (specialization) чи згенерованою функцією (generated function). Процес генерації конкретної функції називається конкретизацією (instantiation). Іншими словами, згенерована функція є конкретним екземпляром шаблонної функції. Тип T, що вказується у кутових дужках, називається параметром шаблону, а тип, що вказується у списку параметрів (наприклад, int) — параметром виклику.

Під час виклику функції (наприклад, max) параметри шаблону визначаються аргументами, що передаються в функцію. Якщо в якості параметрів типу T const& передається два значення int, компілятор робить висновок, що замість T слід підставити int.

Автоматичне перетворення типів в шаблонних функціях не дозволяється. Відповідність типів параметрів і аргументів повинна бути точною.

template <typename T>

void max(T& a, T& b)

...

max(4,5); // вірно

max(4,5.5); // помилка: перший T є int, другий T - double

Існує три способи виправлення цієї помилки.

1. Привести обидва аргументи до одного типу:

max (static\_cast <double>(4), 5.5);

2. Вказати тип T явно

max <double>(4, 5.5);

3. Задати різні типи параметрів шаблонів.

# Шаблонні функції: перевантаження шаблонної функції, використання стандартних параметрів.

Для того, щоб перевантажити специфікацію шаблонної функції, достатньо створити ще одну версію шаблона, що відрізняється від інших своїм списком параметрів.

template <class T> void f(T a) // перша версія шаблонної функції f()

{ cout << "Inside f(T a)\n"; }

template <class T, class Y> void f(T a, Y b) // друга версія шаблонної функції f()

{ cout << "Inside f(T a, Y b)\n"; }

int main()

{ f(10); // виклик функції f(T)

f(10, 20); // виклик функції f(T, Y)

}

# Шаблонні класи: поняття, особливості задання і використання.

Шаблонні класи виявляються корисними, якщо логіка класу не залежить від типу даних. Наприклад, до черг, що складаються з цілих чисел або символів, можна застосовувати один і той самий алгоритм.

Оголошення шаблонного класу має наступний вид.

template <class T> class ім’я\_класу

{ . . . }

Тут параметр T задає тип даних, що уточнюється при створенні об’єкта класу. За необхідності можна визначити декілька шаблонних типів, використовуючи список імен, розділених комами.

Конкретний об’єкт шаблонного класу створюється за допомогою наступної синтаксичної конструкції.

ім'я\_класу <тип> ім'я\_об'єкта;

# Спеціалізація шаблонних класів: різновиди, особливості реалізації.

Як і при використанні шаблонних функцій, можна створити явну спеціалізацію шаблонного класу. Для цього застосовується конструкція template<>.

template <class T> class myclass

{ T x;

public:

myclass(T a)

{ cout << "Шаблонний клас myclass\n";

x = a;

}

T getx() { return x; }

};

template <> class myclass <int> // явна спеціалізація для типу int

{ int x;

public:

myclass(int a)

{ cout << "Спеціалізація myclass<int>\n";

x = a \* a;

}

int getx() { return x; }

};

int main()

{ myclass <double > d(10.1);

cout << "double: " << d.getx() << "\n\n";

myclass i(5);

cout << "int: " << i.getx() << "\n";

getch();

}

Зверніть увагу на наступну рядок програми.

template <> class myclass <int>

Вона повідомляє компілятору, що створюється явна цілочисельна спеціалізація шаблонного класу myclass. Така ж синтаксична конструкція застосовується і для будь-якої іншої спеціалізації класу.

Явна спеціалізація класу розширює можливості шаблонних класів, оскільки вона дозволяє обробляти одну чи дві особливі ситуації, надаючи компілятору можливість автоматично генерувати інші спеціалізації. Якщо програмі буде потрібно занадто багато спеціализацій, можливо, варто взагалі відмовитися від шаблонних класів.

# Виняткові ситуації: поняття, загальний механізм обробки.

В подальшому будемо називати винятковою ситуацією будь-яку подію, що вимагає особливої обробки. При цьому зовсім неважливо, чи є ця подія фатальною чи простою помилкою. Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається обробкою виняткової ситуації. Ця задача покладається на оброблювача виняткової ситуації.

Обробка виняткових ситуацій у мові С++ є об’єктно-орієнтованою. Це значить, що виняткова ситуація є об'єктом, що генерується при виникненні незвичайних умов, передбачених програмістом, і передається оброблювачу, що її перехоплює. Об'єктом, що описує природу виняткової ситуації, може бути будь-яка сутність — літерал, рядок, об'єкт класу, число і т.ін. Не слід думати, що виняткова ситуація обов'язково повинна бути об'єктом якого-небудь класу.

# Механізм обробки виняткових ситуацій: засоби реалізації виключень, їх загальна характеристика.

В основі обробки виняткових ситуацій у мові С++ лежать три ключових слова: try, catch і throw. Якщо програміст підозрює, що визначений фрагмент програми може спровокувати помилку, він повинен заключити цю частину коду в блок try. Необхідно мати на увазі, що зміст помилки (за винятком стандартних ситуацій) визначає сам програміст. Це значить, що програміст може задати будь-яку умову, що приведе до створення виняткової ситуації. Після цього необхідно вказати, у яких умовах варто генерувати виняткову ситуацію. Для цієї мети призначене ключове слово throw. І нарешті, виняткову ситуацію потрібно перехопити і обробити в блоці catch. Ось як виглядає ця конструкція:

try

{ // тіло блоку try if(умова)throw виняткова\_ситуація

}

catch(тип1 аргумент)

{ // тіло блоку catch

}

catch(тип2 аргумент)

{ // тіло блоку catch

}

. . .

catch(типN аргумент)

{ // тіло блоку catch

}

Розмір блоку try не обмежений. У нього можна помістити як один оператор, так і цілу програму. Один блок try можна зв'язати з довільною кількістю блоків catch. Оскільки кожен блок catch відповідає окремому типу виняткової ситуації, програма сама визначить, який з них виконати. У цьому випадку інші блоки catch не виконуються. Кожен блок catch має аргумент, що приймає визначене значення. Цей аргумент може бути об'єктом будь-якого типу.

Якщо програма виконана правильно і у блоці try не виникло жодної виняткової ситуації, усі блоки catch будуть проігноровані. Якщо в програмі виникла подія, яку програміст вважає небажаною, оператор throw згенерує виняткову ситуацію. Для цього оператор throw повинен знаходитися усередині блоку try або усередині функції, викликуваної усередині блоку try.

# Особливості обробки виняткових ситуацій: тотальне перехоплення, список дозволених виняткових ситуацій, непередбачені виняткові ситуації.

Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехо-плення і обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort().

Іноді ретельна деталізація виняткових ситуацій не потрібна. Наприклад, у попе-редніх прикладах їх обробка проводилася майже однаково, за винятком супутніх повідомлень про помилки. Отже, було б зручно, якби блок catch можна було настроїти на будь-яку виняткову ситуацію. Зробити це дозволяє наступна конструкція:

catch(...)

{ // перехоплення усіх виняткових ситуацій

}

Трикрапка в дужках означає, що блок catch здатний перехопити і обробити будь-яку виняткову ситуацію.

Оголошуючи функцію, можна перелічити типи виняткових ситуацій, які їй дозволено генерувати. Типи виняткових ситуацій, не включені в список, функція генерувати не зможе. Спроба порушити це обмеження приведе до негайного припинення роботи програми за допомогою ланцюжка викликів стандартних функцій: unexpected() — abort(). Якщо список порожній, функція взагалі не повинна генерувати ніяких виняткових ситуацій.

Для оголошення списку дозволених виняткових ситуацій використовується наступна синтаксична конструкція:

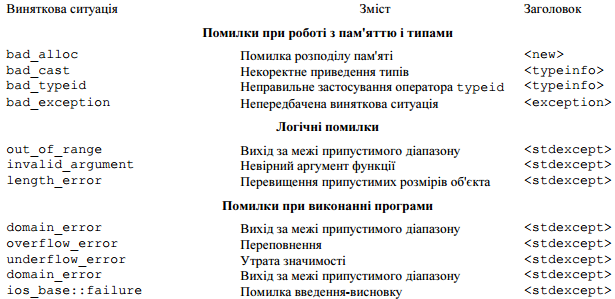
тип\_ результату ім'я\_функції(аргументи) throw(виняткові ситуації)

{ // ...

}

# Стандартні виняткові ситуації: категорії стандартних класів помилок, їх загальна характеристика.

У мові С++ передбачено вісім стандартних виняткових ситуацій, що генеруються операторами і конструкторами стандартних класів.



Ці класи утворять ієрархію, коренем якої є клас exception.

