Лекція 8 : ООП

Об’єктно-орієнтоване програмування. Абстракція, методо та члени класу.
Створення класів за допомогою struct та class. Інкапсуляція. Різниця між public та private.
Конструктори та деструктори.
Статичні методи та члени класу.

Об’єктно-орієнтований підхід полягає в наступному наборі **основних принципів**:

- Все є об’єктами.
- Всі дії та розрахунки виконуються шляхом взаємодії (обміну даних) між об’єктами, при якій один об’єкт потребує, щоб інший об’єкт виконав деяку дію. Об’єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення - це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при виконанні дії.
- Кожен об’єкт має незалежну пам’ять, яка складається з інших об’єктів.
- Кожен об’єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об’єктів.
- У класі задається поведінка (функціональність) об’єкта. Таким чином усі об’єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.
- Класи організовані у єдину деревовидну структуру з загальним корінням, яка називається ієрархією успадкування. Пам’ять та поведінка, зв’язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Таким чином, програма являє собою набір об’єктів, що мають стан та поведінку. **Об’єкти** взаємодіють за допомогою використання **повідомлень**. Будується ієрархія об’єктів: програма в цілому — це об’єкт, для виконання своїх функцій вона звертається до об’єктів, що містяться у ньому, які у свою чергу виконують запит шляхом звернення до інших об’єктів програми. Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об’єкт трансформує запит у повідомлення до стандартних системних об’єктів, що даються мовою та середовищем програмування. Стійкість та керованість системи забезпечуються за рахунок чіткого розподілення відповідальності об’єктів (за кожну дію відповідає певний об’єкт), однозначного означення інтерфейсів міжоб’єктної взаємодії та повної ізольованості внутрішньої структури об’єкта від зовнішнього середовища (інкапсуляції).
Основною метою створення мови програмування C ++ є додавання об'єктно орієнтовних властивостей до мови програмування C. ***Таким чином, класи є центральною особливістю мови C ++, що підтримує об'єктно-орієнтоване програмування і часто називаються користувацькими типами***.

Тобто **клас** - це ***тип даних, що визначений користувачем***, має **окреслену структуру даних**, з яких складається (тобто може складатись зі стандартних типів, а може включати й інші типи даних, що визначені користувачем) та **перелік функцій,** які можуть виконувати **дії всередині класу** та *окреслено коло функцій інших класів*, що мають можливість працювати з даним типом.
Клас використовується для визначення форми об'єкта, і він поєднує представлення даних і способи маніпулювання цими даними в один акуратний пакет. Дані в класі називаються членами класу (class members). Функції в класі звуться методами класу (class methods, function methods).

Визначення класів C ++

Коли ви визначаєте клас, ви визначаєте план для нового типу даних. Це насправді не визначає ніяких даних, але воно визначає ім'я класу та його структуру, тобто з чого буде складатися об'єкт класу і які операції можна виконувати на такому об'єкті.
Визначення класу починається з класу ключового слова, за яким йде ім'я класу; і тіло класу, укладене парою фігурних дужок. Визначення класу повинно супроводжуватися крапкою з комою або списком декларацій. Наприклад, ми визначили тип даних Box за допомогою ключового слова class наступним чином:

```
class Box {
  public: // загальний доступ до челнів класу  (пояснення - далі)
    double length;   // Довжина коробки
    double breadth;  // Ширина коробки
    double height;   // Висота коробки
};// декларація класу завершується крапкою з комою
```

Насправді на Сі++ клас можна визначити й за допомогою ключового слова struct і новий сенс цього ключового слова - ще одна відмінність Сі++ від Сі.

```
struct Box {
  public: // загальний доступ насправді в структурах є й по замовченню
    double length;
    double breadth;
    double height;
};// декларація класу завершується крапкою з комою
```

Примітка. В структурах насправді рівень доступу **public** є й по замовченню – його можна явно не писати, а ось в класах – це обов’язково (там де потрібно).

Ключове слово **public** визначає атрибути доступу членів класу, що слідують за ним. ***Загальнодоступний член*** можна отримати ***за межами класу*** в будь-якому місці в межах об'єкта класу. Ви також можете вказати члени класу як приватні або захищені.

Визначення об'єктів C ++

Клас надає макет структури для об'єктів, тому в основному об'єкт створюється з класу. Конкретний екземпляр класу зветься об’єктом. Оголошуються об'єкти класу точно такою ж декларацією, що оголошуються змінні базових типів. Наступні заяви оголошують два об'єкти класу Box:

```
Box Box1; // Оголошуємо Box1 типу Box
Box Box2; // Оголошуємо Box2 типу Box
```

Обидва об'єкти Box1 і Box2 матимуть власну копію членів даних.

Таким чином, клас - це **абстрактний тип даних**. За допомогою класу описується деяка сутність (її характеристики і можливі дії). Наприклад, клас може описувати студента, автомобіль і так далі. Описавши клас, ми можемо створити його примірник - об'єкт. **Об'єкт** - це вже **конкретний** представник класу.

Для реалізації концепцій об'єктно-орієнтовного програмування (ООП) виділяють наступні поняття:

- Абстракція
- Інкапсуляція
- Наслідування
- Поліморфізм

Абстракція - дозволяє виділяти з деякої сутності тільки необхідні характеристики і методи, які повною мірою (для поставленої задачі) описують об'єкт. Наприклад, створюючи клас для опису студента, ми виділяємо тільки необхідні його характеристики, такі як ПІБ, номер залікової книжки, група. Тут немає сенсу додавати поле вагу або ім'я його кота / собаки тощо

Інкапсуляція - дозволяє приховувати внутрішню реалізацію. У класі можуть бути реалізовані внутрішні допоміжні методи (функції-члени), поля (члени класу), до яких доступ для користувача необхідно заборонити, тут і використовується інкапсуляція.

Наслідування (спадковість) - дозволяє створювати новий клас на базі іншого. Клас, на базі якого створюється новий клас, називається *базовим* (або *батьківським*), а той, що базується новий клас – *спадкоємцем* (*наслідником*). Наприклад, є базовий клас тварина. У ньому описані загальні характеристики для всіх тварин (клас тварини, вага). На базі цього класу можна створити класи спадкоємці: Собака, Слон зі своїми специфічними властивостями. Всі властивості і методи базового класу при спадкуванні переходять в клас спадкоємець.

Поліморфізм - це здатність об'єктів з одним інтерфейсом мати різну реалізацію. Наприклад, є два класи, Круг і Квадрат. У обох класів є метод GetSquare (), який

обчислює і повертає площу. Але площа кола і квадрата обчислюється по-різному, відповідно, реалізація одного і того ж методу різна.

Абстракція

Всі програми C ++ складаються з наступних двох основних елементів:

• Висловлювання програми (код) - це частина програми, яка виконує дії, і вони називаються функціями.

• Дані програми - це інформація про програму, на яку впливають функції програми.

Абстракція даних є технікою програмування (і дизайну), яка спирається на поділ інтерфейсу і реалізації.

Візьмемо один реальний приклад телевізора, який можна вмикати і вимикати, змінювати канал, регулювати гучність і додавати зовнішні компоненти, такі як динаміки, відеомагнітофони та DVD-програвачі, але ви не знаєте його внутрішніх деталей, ви не знаєте, як він отримує сигнали по повітрю або через кабель, як він їх переводить, і, нарешті, відображає їх на екрані.

Таким чином, можна сказати, що телебачення чітко відокремлює його *внутрішню реалізацію* від *зовнішнього інтерфейсу*, і ви можете грати з його інтерфейсами, як кнопка живлення, перемикач каналів і регулятор гучності, не маючи ніяких знань про його внутрішні елементи.

У C ++ класи надають великий рівень абстракції даних. Вони забезпечують достатньо загальнодоступних методів для зовнішнього світу, щоб грати з функціональністю об'єкта і маніпулювати даними об'єктів, тобто, не знаючи, як клас був реалізований внутрішньо.

Наприклад, ваша програма може зробити виклик функції sort (), не знаючи, який алгоритм фактично використовує для сортування заданих значень. Насправді, основна реалізація функцій сортування може змінюватися між випусками бібліотеки, і поки інтерфейс залишається незмінним, виклик функції буде працювати.

У C ++ використовуються класи для визначення власних ***абстрактних типів даних (ADT)***. Ви можете використовувати об'єкт **cout** класу **ostream** для передачі даних у стандартний вивід, як наприклад -

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

int main() {
  cout << "Hello C++" <<endl;
  return 0;
}
```

Тут вам не потрібно розуміти, як cout відображає текст на екрані користувача. Потрібно лише знати загальнодоступний інтерфейс, а базову реалізацію "cout" можна змінювати.

У C ++ використовуються ***різні рівні доступу*** для визначення абстрактного інтерфейсу для класу. Клас може містити нуль або більше міток доступу:

- Члени, позначені загальнодоступною міткою (public), які ***доступні*** для *всіх* частин програми та ***інших*** *класів*. Вигляд абстракції даних типу визначається його загальнодоступними членами.
- Члени, визначені приватною або захищеною міткою(private, protected), ***не доступні*** для коду, який використовує клас. Приватні члени приховують реалізацію від коду, який використовує цей тип.

Немає обмежень щодо частоти появи мітки доступу. Кожна мітка доступу визначає рівень доступу наступних визначень членів. Зазначений рівень доступу залишається в силі доти, доки не з'явиться наступна мітка доступу або не буде закриваюча права дужка тіла класу.

Переваги абстракції даних

Абстракція даних дає дві важливі переваги:

- Внутрішні елементи класу захищені від випадкових помилок на рівні користувача, які можуть пошкодити стан об'єкту.
- Реалізація класу може з часом змінюватися у відповідь на зміну вимог або повідомлень про помилки, не вимагаючи зміни коду рівня користувача.

Визначаючи член тільки в приватному розділі класу, автор класу може вносити зміни до даних непомітно від інших користувачів класу. Якщо реалізація змінюється, необхідно лише змінювати методи які залежать від змінених методів. В іншому ж випадку якщо дані є загальнодоступними, будь-яка функція, яка безпосередньо отримує доступ до даних старого представника, може бути порушена.

Приклад абстракції даних

Будь-яка програма C ++, де реалізується клас з загальнодоступними та приватними членами, є прикладом абстракції даних. Розглянемо наступний приклад

```
#include <iostream>
using namespace std;
class Adder {
  public:
    // Конструктор
    Adder(int i = 0) {
      total = i;
    }
    // метод - інтерфейс
    void addNum(int number) {
      total += number;
```

```
    }
    // метод - інтерфейс
    int getTotal() {
      return total;
    };

  private:
    // дані, що приховані від користувача
    int total;
};

int main() {
  Adder a;
  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат –

```
Усього 60
```

Над класом додаються числа разом, і повертає суму. Публічні члени - addNum і getTotal є інтерфейсами для зовнішнього світу, і користувачеві потрібно знати їх, щоб використовувати клас. Загальна кількість учасників - це те, про що користувач не повинен знати, але він необхідний для правильного функціонування класу.

Стратегія проектування: Абстракція розділяє код на інтерфейс і реалізацію. Таким чином, при розробці вашого компонента, ви повинні тримати інтерфейс незалежним від реалізації, так що якщо ви зміните базову реалізацію, то інтерфейс залишиться незмінним. У цьому випадку будь-які програми, які використовують ці інтерфейси не впливатимуть на роботу класів і не потребуватимуть перекомпіляції разом з ними.

Інкапсуляція

Інкапсуляція - це концепція об'єктно-орієнтованого програмування, яка пов'язує разом дані та функції, які маніпулюють даними, і яка зберігає безпеку від зовнішніх перешкод і неправильного використання. Інкапсуляція даних призвела до важливої концепції сховища даних.

Інкапсуляція даних є механізмом комплектації даних, а функції, які використовують їх і абстракцію даних, є механізмом викриття тільки інтерфейсів і приховування деталей реалізації від користувача.
C ++ підтримує властивості інкапсуляції і даних, що ховаються за допомогою створення визначених користувачем типів, які називаються класами. Отже, клас може містити приватні, захищені та загальнодоступні учасники. За замовчуванням всі елементи, визначені в класі за допомогою ключового слова class, є приватними. Наприклад:

```
class Box {
  public: // загальнодоступні члени та методи
    double getVolume(void) {
      return length * breadth * height;
    }

  private:  // захищені члени та методи
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
```

Члени класу length, breadth, height - мають мітку доступу **private** - приватну. Це означає, що до них можуть звертатися лише інші члени класу Box, а не будь-яка інша частина вашої програми. Це є одним із способів досягнення інкапсуляції.
Щоб зробити частину класу загальнодоступною (тобто доступною для інших частин вашої програми), ви повинні оголосити їх після ключового слова public. Всі змінні або функції, визначені після публічного специфікатора, доступні для всіх інших функцій вашої програми.
Створення одного класу другом іншого викриває деталі реалізації та зменшує інкапсуляцію. Ідеальним є збереження якомога більшої кількості деталей кожного класу від усіх інших класів.

Приклад інкапсуляції даних

Будь-яка програма C ++, де реалізується клас з загальнодоступними і приватними членами, є прикладом інкапсуляції даних і абстракції даних. Розглянемо наступний приклад

```
class TimeH{
   /* private :  Ключове слово private можна не використовувати – це специфікатор по замовченню */
   //захищені члени
   int hours;
   int minutes;
   //цей метод доступний лише з цього класу
   int getTotalMinutes() {
      return hours*60 + minutes;
```

```
  }

public: // публічні члени та методи
  // Конструктор
  TimeH(int h=0, int m=0) {
    setTime(h,m);
  }
  //метод для виведення часу - загальнодоступний
  void show() {
    cout<<"H:"<<hours<<":"<<minutes<<" ";
  }
}; // декларація класу завершується крапкою з комою
```

Даний клас додає числа разом і повертає суму. Публічні члени – конструктор TimeH і show() є інтерфейсами для зовнішнього світу, і користувач повинен знати їх, щоб використовувати клас. Закритий приватний член - це те, що приховано від зовнішнього світу, але необхідне, щоб клас працював належним чином.

Стратегія проектування

Більшість з нас навчилися робити особистим класом приватні за замовчуванням, якщо ми дійсно не повинні їх викривати. Це просто хороша інкапсуляція.

Це найчастіше застосовується до членів даних, але застосовується однаково до всіх членів, включаючи віртуальні функції.

Доступ до членів даних

Доступ до відкритих членів даних об'єктів класу можна отримати за допомогою оператора прямого доступу (.). Давайте спробуємо наступний приклад, щоб зробити все зрозумілим -

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

class Box {
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};

int main() {
  Box Box1;       // Декларуємо Box1 класу Box
  Box Box2;       // Декларуємо Box2 of typ класу Box
  double volume = 0.0;     // Store the volume of a box here
```

```
  // box 1 визначаємо
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;

  // box 2 визначаємо
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;

  // виводимо обєм box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;

  // виводимо обєм box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

Результат:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Важливо відзначити, що приватним і захищеним членам неможливо отримати доступ до членів класу безпосередньо за допомогою оператора прямого доступу (.). Ми дізнаємося, як можна отримати доступ до приватних і захищених членів.

Методи класів і об'єктів

Функція-член (метод) класу - це функція, яка має своє визначення або свій прототип у визначенні класу, як і будь-яка інша змінна. Вона діє на будь-який об'єкт класу, членом якого він є, і має доступ до всіх членів класу для цього об'єкту.

Візьмемо раніше визначений клас для доступу до членів класу, використовуючи функцію-член, замість прямого доступу до них

```
class Box {
  public:
    double length;       // Length of a box
    double breadth;       // Breadth of a box
    double height;       // Height of a box
    double getVolume(void);// Returns box volume
};
```

Функції-члени можуть бути визначені в межах визначення класу або окремо за допомогою оператору визначення меж області (::) . Визначення методу в межах визначення класу оголошує функцію inline, навіть якщо ви не використовуєте вбудований специфікатор. Тому або ви можете визначити функцію getVolume (), як показано нижче

```
class Box {
  public:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box

    double getVolume(void) {
      return length * breadth * height;
    }
};
```

Можна також визначити метод getVolume за межами оператору визначення меж **(scope resolution operator) (::)** наступним чином:

```
double Box::getVolume(void) {
  return length * breadth * height;
}
```

Тут важливим є те, що ви повинні використовувати ім'я класу тільки перед **оператором ::.** Функція-член буде викликана за допомогою оператора крапки (.) На об'єкті можна керувати даними, пов'язаними з цим об'єктом, лише наступним чином:

```
Box myBox;        // Create an object
myBox.getVolume();  // Call member function for the object
```

Приклад.

```
#include <iostream>

using namespace std;

class Box {
  public:  // декларація членів класу
    double length;       // Length of a box
    double breadth;       // Breadth of a box
    double height;       // Height of a box

    // Декларація методів
    double getVolume(void);
    void setLength( double len );
```

```
    void setBreadth( double bre );
    void setHeight( double hei );
};

// Визначення методів
double Box::getVolume(void) {
  return length * breadth * height;
}

void Box::setLength( double len ) {
  length = len;
}
void Box::setBreadth( double bre ) {
  breadth = bre;
}
void Box::setHeight( double hei ) {
  height = hei;
}
// Головна функція
int main() {
  Box Box1;             // Декларація Box1 типу Box
  Box Box2;             // тобто декларація обєкту Box2 класу Box
  double volume = 0.0;     // змінна для зберігання згачення обєму

  // box1 специфікація
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // отримаємо обєм box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // обєм box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;
```

```
  return 0;
}
```
Результат:
Volume of Box1 : 210
Volume of Box2 : 1560

Приховування даних є однією з важливих особливостей об'єктно-орієнтованого програмування, яка дозволяє запобігти безпосередньому доступу до програми внутрішнього представлення типу класу. Обмеження доступу до членів класу визначається позначеними загальнодоступними, приватними та захищеними розділами в тілі класу. Ключові слова **public**, **private** і **protected** називаються **специфікаторами доступу**.

Клас може мати кілька загальнодоступних, захищених або приватних методів та членів. Кожна секція залишається в силі доти, доки не з’явиться будь-яка інша мітка або закриється остання фігурна дужка тіла класу. Стандартний доступ для членів і класів є приватним (для класу, що визначений ключовим словом class) та публічним (для класу, що визначений ключовим словом struct) .

```
class Base {
  public:
    // загальнодоступні члени та методи
  protected:
  // приховані члени та методи
  private:
  // захищені члени та методи
};
```

Загальнодоступний член доступний з будь-якого місця поза класом, але в межах програми. Ви можете встановити та отримати значення загальнодоступних змінних без будь-якої функції-члена, як показано в наступному прикладі

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

class Line {
  public: // декларація членів та методів
    double length;
    void setLength( double len );
    double getLength( void );
};

// Визначення  методів
double Line::getLength(void) {
  return length ;
}
```

```
void Line::setLength( double len) {
  length = len;
}

int main() {
  Line line;
  // встановити довжину за допомогою публічного  методу
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  // встановити довжину за допомогою доступу до публічного члену
  line.length = 10.0; // OK: because length is public
  cout << "Length of line : " << line.length <<endl;
  return 0;
}
```

Результат роботи коду:

```
Length of line : 6
Length of line : 10
```

До приватної змінної чи функції члена не можна звертатися, або навіть переглядати за межами класу. До приватних учасників можуть звертатися лише функції класу та друга.

За замовчуванням всі члени класу будуть приватними, наприклад, у наступній програмі ширина (width) є приватним членом класу. Отримати доступ до неї з-за меж класу можна лише за допомогою публічних методів класу

```
void setWidth( double wid );
```

та

```
double getWidth( void );
```

Приклад.

```
#include <iostream>
using namespace std;
class Box {
  public: // декларація членів та методів
    double length;
    void setWidth( double wid );
    double getWidth( void );
  private:
    double width;
};

// Визначення методів
```

```
double Box::getWidth(void) {
  return width ;
}

void Box::setWidth( double wid ) {
  width = wid;
}
int main() {
  Box box;
     // встановити довжину за допомогою публічного  методу
  box.length = 10.0; // OK: because length is public
  cout << "Length of box : " << box.length <<endl;
  // спроба встановити ширину
  // box.width = 10.0; // Помилка: width is private!!!
  box.setWidth(10.0);  // можна використати публічну функцію
  cout << "Width of box : " << box.getWidth() <<endl;

  return 0;
}
```

Результат роботи коду:

```
Length of box : 10
Width of box : 10
```

Конструктор класу

Конструктор класу - це **спеціальна функція-член класу**, що виконується, коли ми **створюємо нові об'єкти цього класу**.

Конструктор *матиме* точно таку саму ***назву***, що і ***клас***, і він взагалі ***не має типу повернення***, навіть не має значення void. Конструктори можуть бути дуже корисними для встановлення початкових значень для певних змінних-членів.

Наступний приклад пояснює концепцію конструктора.

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

class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line();  // Це конструктор
  private:
    double length;
```

```
};

// Визначимо якусь дію всередині конструктора
Line::Line(void) {
  cout << "Object is being created" << endl;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

int main() {
  Line line; // конструктор створює нам
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  return 0;
}
```

Результат роботи коду:

```
Object is being created
Length of line : 6
```

Примітка. Даний конструктор насправді не має особливої потреби створювати, оскільки порожній непараметризований конструктор автоматично створюється при декларації класу (***конструктор за замовчуванням***).

Параметризований конструктор

Конструктор за замовчуванням не має жодного параметра, але, якщо потрібно, конструктор може мати параметри. Це допоможе вам призначити початкове значення об'єкту під час його створення, як показано в наступному прикладі

```
#include <iostream>

using namespace std;
class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line(double len);  // Це також конструктор
  private:
    double length;
```

```
};

// Визначення конструктора та інших функцій
Line::Line( double len) {
  cout << "Object is being created, length = " << len << endl;
  length = len;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

int main() {
  Line line(10.0); // виклик конструктору
   // отиримуємо встановлену конструктором довжину
  cout << "Length of line : " << line.getLength() <<endl;
  // змінюємо довжину
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  return 0;
}
```

Результат роботи програми:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Альтернативний метод визначення параметризованого конструктору:

```
Line::Line( double len): length(len) {
  cout << "Object is being created, length = " << len << endl;
}
```

Цей запис рівносильний

```
Line::Line( double len) {
  cout << "Object is being created, length = " << len << endl;
  length = len;
}
```

Тобто для класу C, якщо у нас є наприклад члени класу X, Y, Z, etc., ми можемо використовувати наступний синтаксис: відокремлення полів та ініціалізація їх в круглих дужках:

```
C::C( double a, double b, double c): X(a), Y(b), Z(c) {
```

```
  ....
}
```

Примітка 1: Ініціалізування об’єкту при наявності параметризованого конструктору. У випадку наявності параметризованого конструктора можна використати ініціалізацію об’єкту за допомогою фігурних дужок:

```
Line pryama1{12}, pryama2{25};
C object_c{1,2,3};
```

Примітка 2: В одному класі може бути визначено декілька конструкторів (кожен з різними аргументами).

Примітка 3: Якщо в класі визначений параметризований конструктор, то конструктора за замовченням вже немає, тому якщо він все ж таки теж потрібен (наприклад для ініціалізацію порожнього масиву з об’єктів нашого класу),потрібно додати в клас і непараметризований конструктор.

Деструктор класу

Деструктор - це **спеціальна функція-член класу**, що виконується, коли об'єкт його класу виходить за межі області або коли вираз видалення (delete) застосовується до вказівника на об'єкт цього класу.
Деструктор матиме точно таку саму **назву**, що і **клас**, лише до цієї назви додається **символ тильди (~),** і він ***не може ні повертати значення, ні прийняти будь-які параметри***. Деструктор може бути дуже корисним для звільнення ресурсів перед виходом з програми, наприклад, закриття файлів, вивільнення пам'яті тощо.
Наступний приклад пояснює концепцію деструктора :

```
#include <iostream>
using namespace std;
class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line();   // Декларація конструктору
    ~Line();  // Декларація деструктору
  private:
    double length;
};

// Визначення методів, зокрема конструктору та деструктору
Line::Line(void) {
  cout << "Object is being created" << endl;
```

```
}
Line::~Line(void) {
  cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

int main() {
  Line line;
  // встановити довжину
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  return 0;
}
```

Результат роботи програми:

```
Object is being created
Length of line : 6
Object is being deleted
```

Конструктор копіювання

Конструктор копіювання - це **конструктор**, який створює об'єкт, ***ініціалізуючи*** його ***об'єктом того ж класу***, який був створений раніше. Конструктор копіювання використовується для

- Ініціалізації одного об'єкту з іншого того ж типу.
- Копіювання об'єкту, щоб передати його як аргумент функції.
- Копіювання об'єкту, щоб повернути його з функції.

Якщо конструктор копіювання не визначений в класі, сам компілятор визначає один. Якщо клас має змінні вказівники і має деякі динамічні виділення пам'яті, то необхідно мати конструктор копіювання. Найбільш поширеною формою конструктора копіювання є така

```
classname (const classname &obj) {
  // тіло конструктору
}
```

Приклад

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

```
class Line {
  public:
    int getLength( void );
    Line( int len );          // звичайний конструктор
    Line( const Line &obj);  // конструктор копії
    ~Line();                  // деструктор
  private:
    int *ptr;
};

// Визначення методів, зокрема конструкторів та деструктору
Line::Line(int len) {
  cout << "Normal constructor allocating ptr" << endl;
  // виділяємо пам'ять під вказівник
  ptr = new int;
  *ptr = len;
}

Line::Line(const Line &obj) {
  cout << "Copy constructor allocating ptr." << endl;
  ptr = new int;
  *ptr = *obj.ptr; // копіювання
}

Line::~Line(void) { // деструктор звільняє память
  cout << "Freeing memory!" << endl;
  delete ptr;
}

int Line::getLength( void ) {
  return *ptr;
}

void display(Line obj) {
  cout << "Length of line : " << obj.getLength() <<endl;
}
int main() {
  Line line(10);
  display(line);
  return 0;
}
```

Результат роботи:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Спробуємо використати цей код

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

class Line {
  public:
    int getLength( void );
    Line( int len );           // конструктор
    Line( const Line &obj);  // конструктор копіювання
    ~Line();                  // деструктор

  private:
    int *ptr;
};

// Визначення методів, зокрема конструкторів та деструктору
Line::Line(int len) {
  cout << "Normal constructor allocating ptr" << endl;
  // виділяємо пам'ять під вказівник
  ptr = new int;
  *ptr = len;
}

Line::Line(const Line &obj) {
  cout << "Copy constructor allocating ptr." << endl;
  ptr = new int;
  *ptr = *obj.ptr; // копіювання
}

Line::~Line(void) {
  cout << "Freeing memory!" << endl;
  delete ptr;
}

int Line::getLength( void ) {
```

```
  return *ptr;
}

void display(Line obj) {
  cout << "Length of line : " << obj.getLength() <<endl;
}

int main() {
  Line line1(10); // Конструктор
 Line line2 = line1; // Це також конструктор- копіювання
  display(line1);
  display(line2);
  return 0;
}
```

Результат роботи:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Закон трьох

"Закон трьох" насправді не є законом, а скоріше орієнтиром: **якщо клас потребує явно заявленого конструктора копіювання, оператора присвоєння копії або деструктора, то зазвичай він потребує всіх трьох.**

Існують винятки з цього правила (або, скоріше, уточнення). Наприклад, іноді деструктор явно оголошується тільки для того, щоб зробити його віртуальним; у цьому випадку не обов'язково потрібно оголошувати або реалізовувати оператор копіювання і копіювання конструкторів копіювання.

Більшість класів не повинні оголошувати будь-яку з операцій "великої трійки": класи, які управляють ресурсами, взагалі потребують всіх трьох.

Дружня функція

Дружня функція класу ***визначається за межами цього класу***, але ***має право доступу до всіх приватних і захищених членів*** класу. Незважаючи на те, що прототипи для дружніх функцій з'являються у визначенні класу, дружні функції не є методами класу. Така функція може бути корисною для того щоб працювати з об’єктом класу, та при цьому бути створеною за межами класу.

Дружня функція може бути ***функцією***, ***шаблоном функції*** або ***функцією-членом(методом)***, або ***шаблоном класу*** або ***класом***, у цьому випадку весь клас і всі його члени є дружними.
Щоб оголосити функцію другом класу, передуйте прототипу функції у визначенні класу ключовим словом friend наступним чином

```
class Box {
  double width;

  public:
    double length;
    friend void printWidth( Box box );
    void setWidth( double wid );
};
```

Для того, щоб визначити методи класу ClassTwo дружніми методами класу ClassOne, потрібно помістити наступну декларацію в визначення класу ClassOne :

```
friend class ClassTwo;
```

Приклад.

```
#include <iostream>
using namespace std;
 class Box {
  double width;
  public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};

// визначення методів
void Box::setWidth( double wid ) {
  width = wid;
}

// Примітка: printWidth() не є методом ніякого класу.
void printWidth( Box box ) {
  /* Оскільки printWidth() є другом (friend) класу Box, ця функція має доступ до всіх його членів та методів  */
  cout << "Width of box : " << box.width <<endl;
}
int main() {
  Box box;
  // встановили ширину
```

```
  box.setWidth(10.0);
  // використали дружню функцію для виведення
  printWidth( box );
}
```

Вбудовані функції

С ++ inline функція - потужна концепція, яка зазвичай використовується з класами. Якщо функція є вбудованою, компілятор розміщує копію коду цієї функції в кожній точці, де функція викликається під час компіляції.

Будь-яка зміна вбудованої функції може вимагати перекомпіляції всіх клієнтів функції, оскільки компілятору потрібно буде замінити весь код ще раз, інакше він продовжуватиме стару функціональність.

Щоб вбудувати функцію, розмістіть ключове слово inline перед назвою функції та визначте функцію, перш ніж виклик буде здійснено. Компілятор може ігнорувати вбудований кваліфікатор у випадку, якщо визначена функція міститься на більш ніж одній лінії.

Визначення функції в визначенні класу є вбудованим визначенням функції, навіть без використання вбудованого специфікатора.

Нижче наведено приклад, який використовує функцію inline для повернення максимуму з двох чисел

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

inline int Max(int x, int y) {
  return (x > y)? x : y;
}

int main() {
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
  return 0;
}
```

Результат:

```
Max (20,10): 20
Max (0,200): 200
```

Використання вказівнику this

Сам об'єкт у С ++ має доступ до власної адреси через важливий вказівник **this**, який називається this-вказівником. Цей вказівник є неявним параметром для всіх функцій-членів. Отже, всередині методу класу він може використовуватися для посилання на об'єкт, що викликає.

Дружні функції не мають цього покажчика, тому що друзі не є членами класу. Тільки метод класу може викликати цей вказівник.

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

```
class Box {
  public:
    // Конструктор
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }
    int compare(Box box) {
      return this->Volume() > box.Volume();
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Визначили box1
  Box Box2(8.5, 6.0, 2.0);    // Визначили box2

  if(Box1.compare(Box2)) {
    cout << "Box2 is smaller than Box1" <<endl;
  } else {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }

  return 0;
}
```

Результат роботи:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Вказівник на клас

Вказівник на клас C ++ робиться точно так само, як і вказівник на структуру, і для доступу до членів вказівника на клас, який потрібно використовувати оператор доступу ->, так само, як з вказівниками на структури. Крім того, як і для всіх покажчиків, перед його використанням необхідно ініціалізувати безпосередньо сам вказівник.

Спробуємо наступний приклад, щоб зрозуміти поняття вказівника на клас

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

class Box {
  public:
    // Конструктор
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Визначили box1
  Box Box2(8.5, 6.0, 2.0);    // Визначили box2
  Box *ptrBox;                // Визначили вказівник на клас Box

  // Взяли адресу першого обєкту
  ptrBox = &Box1;

  // Доступ до даних через публічні методи
  cout << "Volume of Box1: " << ptrBox->Volume() << endl;
  // Беремо вказівник – інший обєкт
```

```
  ptrBox = &Box2;
  // Отримаємо доступ до нього через вказівник
  cout << "Volume of Box2: " << ptrBox->Volume() << endl;
  return 0;
}
```

Результат:

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

Статичні методи та члени

Клас може визначити статичні елементи з використанням ключового слова static. Коли ми оголошуємо член класу ***статичним***, це означає, що незалежно від того, скільки об'єктів класу створено, ***існує*** тільки ***одна копія статичного члена***.

Статичний член є спільним для ***всіх об'єктів*** класу. Всі статичні дані ініціалізуються до нуля, коли створюється перший об'єкт, якщо немає іншої ініціалізації. Ми не можемо визначити його в середині класу, але його можна ініціалізувати за межами класу, як це зроблено в наступному прикладі, повторно класифікуючи статичну змінну, використовуючи оператор дозволу меж (::) визначивши до якого класу він належить.

Спробуємо наступний приклад, щоб зрозуміти поняття статичних членів:

```
#include <iostream>
using namespace std;
class Box {
  public:
    static int objectCount; // статичний член – лічильник обєктів
    // Конструктор
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      // збільшуємо на одиницю лічильник об’єктів
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }

  private:
```

```
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Ініціалізуємо статичний член класу
int Box::objectCount = 0;

int main(void) {
  Box Box1(3.3, 1.2, 1.5);
  Box Box2(8.5, 6.0, 2.0);
  // Друкуємо загальну кількість обєктів
  cout << "Total objects: " << Box::objectCount << endl;
  return 0;
}
```

Результат:

```
Constructor called.
Constructor called.
Total objects: 2
```

Статичні методи

Оголошуючи ***метод класу статичним***, ви робите його ***незалежним від будь-якого конкретного об'єкта класу***. Статичний метод можна викликати, навіть якщо не існує жодного об'єкта класу, і до статичних функцій можна звертатися, використовуючи тільки ім'я класу і оператор визначення меж області(::). ***Статичний метод*** може отримати ***доступ тільки до статичного члена класу***, *інших статичних методів* і будь-яких інших функцій поза класом.

Статичні методи мають областю дії сам клас, і вони не мають доступ до вказівника класу this. Ви можете використовувати статичний метод, щоб визначити, чи були створені деякі об'єкти класу.

Спробуємо наступний приклад, щоб зрозуміти поняття статичних методів

```
#include <iostream>
 using namespace std;
class Box {
  public:
    static int objectCount;
    // Конструктор
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
```

```
    height = h;
    // збільшуємо на одиницю лічильник об’єктів для кожного обєкту
    objectCount++;
  }
  double Volume() {
    return length * breadth * height;
  }
  static int getCount() { // статичний метод для виведення лічильника
    return objectCount;
  }
 private:
  double length;
  double breadth;
  double height;
};

// Ініціалізація статичного члену потрібна в будь-якому файлі що використовує лічильник
int Box::objectCount = 0;

int main(void) {
 // Друкуємо загальну кількість обєктів
 cout << "Inital Stage Count: " << Box::getCount() << endl;

 Box Box1(3.3, 1.2, 1.5);
 Box Box2(8.5, 6.0, 2.0);

// Друкуємо загальну кількість обєктів
 cout << "Final Stage Count: " << Box::getCount() << endl;

 return 0;
}
```

Результат:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```