Лекція 13. Класи

1. *Навіщо класи мові С++. Термінологія.*
2. *Синтаксис класу. Оголошення полів, методів. Прихована та доступна частини.*
3. *Розбиття визначення класу на файли. Тестування класу. Надсилання повідомлень.*
4. *Конструктори: за замовчуванням, створення, копіювання. Деструктор.*
5. *Порівняння об'єктів, використання вказівника this.*
6. *Статичні та динамічні об'єкти, масиви об'єктів, вказівники на об'єкт.*
7. *Діапазон видимості класу. Вкладені оголошення типів.*

Дуже часто термін *клас* вживають у зв'язку з розглядом питань об'єктно-орієнтованого програмування (ООП). А що таке ООП? Це особливий спосіб програмного моделювання реального світу: об'єкти програми (екземпляри класів) поєднують у собі здатність зберігати стан та демонструвати поведінку, тому є найбільш адекватними моделями реальних сутностей. Це особливий спосіб структурування програм за допомогою побудови та використання ієрархії об'єктних типів – класів. Це особливий спосіб обмеження доступу та приховування даних за допомогою інкапсуляції їх в класі та «обгортання» оболонкою методів доступу. Це ще й особливий спосіб повторного використання коду та удосконалення, розвитку функціональності класів.

Звичайно, всі ці аспекти враховано розробниками С++, є відповідні інструменти мови для реалізації тих чи інших потреб ООП. Та все ж *клас для С++* – це, в першу чергу, досконалий *інструмент оголошення типів користувача*. Типів, що якнайкраще задовольняють особливі потреби користувача, і є такими ж зручними у використанні, як і вбудовані типи: одні і ті ж правила для об'єкта та для цілого числа (наприклад). А що ж таке ТИП? Тип – це, по-перше, певна множина значень і набір пов'язаних з ними операцій, по-друге, реалізація типу визначає певні правила кодування значень цієї множини (зокрема, довжину коду – розмір необхідної пам'яті). Таку інформацію про вбудовані типи зберігає компілятор. Множину значень, набір операцій для нового типу описує програміст за допомогою визначення класу. Компілятор «навчений» будувати для нього спосіб кодування та долучати новий тип до компанії вже визначених.

Саме з такої точки зору – клас як інструмент створення нового типу – варто знайомитися з класами С++. Під час знайомства доведеться звернути увагу на такі важливі моменти: визначення типу (множини значень, операцій, поведінки тощо); оголошення та визначення об'єктів, час життя об'єкта (зокрема, з огляду на клас пам'яті); присвоєння об'єктові, автоматичне перетворення типу.

Розмова буде довгою і не завжди простою, тому спочатку узгодимо термінологію. На жаль, автори вживають у публікаціях, присвячених різним мовам програмування, різні терміни для тих самих понять.

Клас – об'єктний тип; тип, оголошений користувачем, що поєднує дані та методи опрацювання, відкритий для наслідування, для створення на його базі нових типів, є шаблоном для створення екземплярів класу, є «фабрикою» екземплярів.

Об'єкт – екземпляр класу, що має власний набір даних і поділяє методи з іншими екземплярами, змінна об'єктного типу, модель реальної сутності.

Повідомлення – особливий спосіб «звертання» до об'єкта. Опрацюванням повідомлення є відшукання і виконання відповідного методу. У С++ часто говорять про *виклик член-функції*.

Поле даних (екземпляра) – змінна, оголошена в межах класу. В С++ – *член класу, елемент даних*. Поля даних оголошують в класі, але існують вони в екземплярах: кожен об'єкт отримує власний набір полів і використовує їх для зберігання інформації про свій стан.

Змінна класу – статичний член класу в С++. Особливе поле даних, що існує в самому класі, поза будь-яким екземпляром, може зберігати інформацію, спільну для всіх екземплярів класу.

Метод (екземпляра) – член-функція класу в С++, *елемент-функція*. Функція, оголошена в межах класу. Описує поведінку об'єкта. Надає програмний інтерфейс для доступу до даних

об'єкта. Зберігається в словнику методів класу, використовується спільно всіма екземплярами класу. За допомогою методів об'єкт опрацьовує отримані повідомлення. Таким чином, доступ до методів – через надсилання повідомлень об'єктам.

Метод класу – статична член-функція в C++. Метод, виклик якого виконують виключно через ім'я класу. Методи класу працюють зі змінними класу і не можуть звертатися до полів екземпляра.

Конструктор – метод класу, відповідальний за створення та ініціалізацію екземплярів.

Деструктор – метод класу, відповідальний за знищення екземплярів.

Будь-який тип, особливо клас, є абстракцією деякої сутності. Виберіть довільний реальний об'єкт або явище, опишіть його стан та поведінку, відзначаючи найголовніше та опускаючи другорядні деталі, і ви отримаєте непогану базу для написання класу, адже від абстракції до типу – один крок. Синтаксичні правила оголошення класу розглянемо на прикладі.

Розглянемо клас, що моделює пакет акцій [2, ст. 431] у дещо спрощеному варіанті. Припустимо, що один пакет визначають такі дані: назва компанії, кількість акцій у пакеті, ціна однієї акції, загальна вартість пакета. Операції з пакетом обмежимо такими діями: придбати пакет у компанії, купити додаткові акції до того ж пакета, продати пакет, корегувати ціну однієї акції пакета, відобразити дані про пакет.

Тепер можна перейти до визначення класу. Зауважимо, що воно здебільшого складається з двох частин: оголошення класу, в якому описано компоненти даних в термінах елементів даних і загальнодоступний інтерфейс у термінах функцій-елементів, та визначення методів.

Перша спроба оголошення класу *Stock* матиме вигляд:

```
class Stock
{
private:
      char company[30];
      int shares;
      double share_val;
      double total_val;
      void set_tot() { total_val = shares * share_val; }
public:
      void acquire(const char* co, int n, double pr);
      void buy(int num, double price);
      void sell(int num, double price);
      void update(double price);
      void show();
};
```

У оголошенні є чотири поля даних (*company*, *shares*, *share_val*, *total_val*) та шість методів. Видимістю членів класу керують позначки *private* та *public*. (У C++ використовують також *protected*, але про неї мова йтиме пізніше.) За замовчуванням, якщо нема відповідних позначок, члени класу вважаються прихованими – *private*. Як і в інших мовах, прихована і доступна частини незалежні. Відкрита відповідає за інтерфейс, прихована – за реалізацію. Зміни прихованої частини ніяк не впливають на користувачів класу (поки не зачіпають інтерфейсу).

Друга частина – визначення методів. Наведемо приклади деяких з них.

```
void Stock::acquire(const char *co, int n, double pr)
{     strncpy(company, co, 29); company[29] = '\0';
      if (n) shares = n; else shares = 0;
      if (pr>0) share_val = pr; else share_val = -pr;
      set_tot();
}
void Stock::buy(int num, double price)
{     if (num<0) cerr
        <<"Number of shared purchased can't be negative. Transactions is aborted.\n";
      else { shares += num; share_val = price; set_tot(); }
}
```

У таких визначеннях використовують кваліфіковане ім'я методу: *ім'яКласу::ім'я-Методу*. Метод *acquire* моделює початкову купівлю пакета, наступні три методи реалізують відповідно купівлю додаткових акцій, продаж акцій та зміну ціни. Всі ці методи впливають на ціну однієї акції та загальну вартість пакету. Обчислення та збереження загальної вартості оформлено у вигляді окремого методу – *set_tot*. Такий прийом дає змогу, по-перше, зекономити на повторному програмуванні тотожніх дій (у всіх чотирьох випадках), по-друге, що особливо важливо, ґарантувати, що кожного разу обчислення виконуються за тими самими правилами.

Метод *set_tot* не є частиною інтерфейсу, а є лише особливістю прогамної реалізації, тому його оголошено в прихованій частині. Ще одна особливість цього методу – тіло методу визначено одразу в оголошенні класу. *Таке визначення члена-функції автоматично робить її вбудованою*. При бажанні будь-який метод можна зробити вбудованим, якщо його визначення почати ключовим словом *inline*.

Зауважимо також, що методи *acquire*, *buy* (як і *sell*) виконують необхідні перевірки для того, щоб дані, збережені в об'єкті, залишалися коректними.

Тип *Stock* можна використовувати для оголошення змінних так само, як і вбудовані типи:

```
int k;   Stock sally;   Stock* peter = new Stock;   Stock bella[5];
```

Звертання до методів схоже на звертання до членів структури:

```
sally.acquire("IBM",10,300.5);   peter->acquire("AT&T",25,245);
sally.show();                    peter->show();
```

Для того, щоб випробувати новий клас, запишіть у файлі програми: 1) оголошення класу; 2) визначення методів класу; 3) невелику тестову програму, що створює декілька екземплярів класу та взаємодіє з ними через надсилання різноманітних повідомлень (виклики методів). Послідовність пунктів 2), 3) можна поміняти. Проте на практиці ви рідко зустрінете всі три частини в одному файлі. Насправді потрібно виконати наступне: п. 1) записати до заголовкового файла, наприклад, *stock.h*; п. 2) записати до окремого програмного файла, наприклад, *stock.cpp*, заголовковий файл включити до нього за допомогою директиви *include*; тестову програму з п. 3) також розташувати в окремому файлі (*test.cpp*) і включити до нього заголовковий.

Оголошений вище клас забезпечує мінімальну схожість з вбудованими типами. Ми не зможемо, наприклад, визначити об'єкт як записано нижче

```
Stock hot = {"Suzukie Autos Inc.", 200, 50.25};
```

Такий «фокус» ми робили зі звичайними структурами, але тут він не спрацює, бо насправді означає спробу доступу до прихованих полів даних. У нашому коді визначення полів виконував метод *acquire*. Знаємо з попереднього досвіду, що заданням полів новоствореного об'єкта мав би займатися конструктор. Як же тут?

У C++ усі екземпляри створюють конструктори. Навіть, якщо ви не визначили ні одного конструктора (як в нашому прикладі), *компілятор автоматично згенерує конструктор за замовчуванням*. Конструктор за замовчуванням генерується (автоматично і без попередження) для будь-якого класу без явно оголошених конструкторів. Такий неявний автоматичний конструктор *створює екземпляр і зануляє всі його поля*. Іноді цього достатньо, але ми хотіли би мати щось корисніше. Для цього доведеться визначити свій конструктор. У мові C++ конструктором є метод, функція чиє ім'я співпадає з іменем класу і яка не має типу результату (ні імені типу, ні *void*). Фактично типом результату є тип об'єкта – клас, тому ім'я конструктора відіграє подвійну роль: і ім'я функції, і тип результату. Зауважимо, що в тілі конструктора не використовують інструкцію *return* – це заборонено. Метод *acquire* є хорошим кандидатом для перетворення на конструктор, що задає всі значення полів об'єкта (див. повний код прикладу). Зауважимо також, що оголошення будь-якого конструктора відміняє

автоматичну генерацію конструктора за замовчуванням, тому про такий конструктор треба потурбуватися окремо. Тепер оголошення класу набуде вигляду:

```
class Stock
{
private:
     char company[30];
     int shares;
     double share_val;
     double total_val;
     void set_tot() { total_val = shares * share_val; }
public:
     Stock();                                    // конструктор за замовчуванням
     Stock(const char* co, int n=0, double pr=0.); // конструктор - визначення
     ~Stock();                                   // деструктор
     void buy(int num, double price);
     .....
};
```

Тепер визначення об'єкта може мати вигляд

```
Stock sally = Stock("IBM",10,300.5);
```

або

```
Stock sally("IBM",10,300.5);
```

а також

```
Stock * peter = new Stock("AT&T",25,245);
```

Параметри за замовчуванням у конструктора використовують звичайним чином: для того, щоб можна було задавати не всі аргументи в момент виклику. Очевидно, що якщо значення за замовчуванням вказано для *всіх* параметрів, то тоді не потрібно визначати конструктор за замовчуванням: це і зайве, і породжує неоднозначність.

На додачу до звичайних дій з визначення полів об'єкта в тілі конструктора можна передбачити додаткові, які, наприклад, інформували б програміста про створення об'єктів, підраховували їхню кількість тощо.

У новому оголошенні класу з'явився ще один метод – деструктор. Звичайним завданням деструктора є звільнення всіх ресурсів, захоплених конструктором у момент створення чи самим об'єктом впродовж життя. Для такого простого класу як *Stock*, деструкторові просто нічого робити, але ми можемо використати його також з інформативною метою – нехай повідомляє про знищення об'єкта. Зауважимо, що якщо в оголошенні класу не визначено деструктора, компілятор генерує його автоматично. Знищення ж бо об'єктів завжди виконує деструктор.

Клас C++ може мати *довільну кількість конструкторів і тільки один деструктор*. Програма ніколи сама не викликає деструкторів – коли це зробити, вирішує компілятор. Можливо, тому деструктор не може мати параметрів.

Конструктор можна використати для оновлення (перевстановлення) значень полів об'єкта, якщо присвоїти вже наявному об'єктові виклик конструктора його класу. При цьому створиться новий тимчасовий об'єкт, значення його полів скопіюються в поля старого об'єкта. Чи буде при цьому задіяно деструктор, залежить від версії компілятора. Одним з наслідків сказаного є те, що екземпляри одного класу можна присвоювати один одному. При цьому відбувається копіювання значень всіх відповідних полів.

Методи об'єкта мають повний доступ до полів даних, проте не завжди це потрібно, наприклад, метод виведення даних про об'єкт (*show* у нашому класі) не повинен модифікувати цих даних. Для нього об'єкт є константним. Для такого випадку потрібен новий синтаксичний засіб, котрий би ґарантував, що метод не змінюватиме даних. Засіб полягає в тому, що заголовок методу завершують ключовим словом *const*.

```
class Stock
{
private:    .....
public:     .....
     void show() const;
};
```

Так само *const* дописують і у визначенні методу.

Такий метод не лише обіцяє не змінювати звичайні об'єкти, але ще й може працювати з об'єктами-константами:

```
const Stock land = Stock("Kludgehorn Properties");
land.show();          // працює
land.buy(10,100.0);   // не працює!!!
```

Продовжимо роботу з класом *Stock*. Тепер він набуває дедалі привабливішого вигляду, але йому точно бракує "аналітичних здібностей": за наявних засобів ми вже можемо визначати об'єкти, навіть масиви об'єктів, маніпулювати ними, але як знайти найдорожчий пакет акцій (чи найдешевший)? Іншими словами – як порівняти два об'єкти? Клас поки що цього не вміє. Давайте розглянемо варіанти вирішення цього завдання.

Спосіб 1 – зовнішня функція. Очевидно, порівнювати пакети будемо за їх сумарною ціною, тобто за значенням поля *total_val*. Щоб надати зовнішній функції доступ до нього, доповнимо інтерфейс класу та оголосимо саму функцію

```
class Stock
{private:   .....
 public:    .....
     double total() const; { return total_val; }
};
bool greater(const Stock& a, const Stock& b)
{    return a.total()>b.total();     }
```

Функція *greater* дає відповідь на запитання чи $a > b$. Для доступу до числових значень об'єктів вона використовує метод *total*. Мабудь, виконати таку перевірку було би простіше комусь тому, хто має безпосередній доступ до полів об'єкта.

Спосіб 2 – новий метод об'єкта. Доступ до полів мають методи, тому можемо доповнити інтерфейс класу методом порівняння

```
class Stock
{private:   .....
 public:    .....
     bool greaterThan(const Stock& b) const { return total_val > b.total_val; }
};
```

Метод *graterThan* порівнює отримувача з аргументом. Специфікатори *const* вказують, що ніхто з учасників процесу не змінюється. Метод хороший, але ним було би простіше користуватися, якби він повертав не результат порівняння, а більшого з об'єктів.

Спосіб 3 – метод, що повертає об'єкт. Продовжимо наш експеримент

```
class Stock
{private:   .....
 public:    .....
     const Stock& topval(const Stock &s) const
     {      if (s.total_val>total_val) return s;
            else return *this; }
};
```

Метод *topval* повертає посилання, тобто, об'єкт: аргумент *s*, або отримувача **this*. (*this* – це вказівник на отримувача повідомлення, або на об'єкт, що звернувся до методу, присутній в тілі кожного методу, аналог *self* мови Object Pascal).

Тепер у програмі можна писати щось на зразок

```
Stock a,b,top; ..... top = a.topval(b);
```

Методи з двох попередніх двох способів – це чудово, але як же схожість з вбудованими типами? Чи можна порівняти об'єкти з допомогою звичайного оператора >?

Спосіб 4 – перевизначення оператора >. Його ми розглянемо трохи пізніше.

Використання вказівника *this* може виявитися корисним у багатьох випадках. Наприклад, ми могли б дещо удосконалити інтерфейс методів, що виконують дії з пакетом, і замість відсутнього типу *void* використати посилання на отримувача.

```
class Stock
{private:   .....
 public:    .....
      Stock& buy(int num, double price);
      Stock& sell(int num, double price);
};
Stock& Stock::buy(int num, double price)
{     if (num<0) cerr
        <<"Number of shared purchased can't be negative. Transactions is aborted.\n";
      else { shares += num; share_val = price; set_tot();
      } return *this;
}
```

Тепер декілька дій з об'єктом можна об'єднувати в один ланцюжок:

```
Stock a,b; ..... a.buy(20,150.5).show(); b.buy(300,100).sell(300,150);
```

Ми вже згадували раніше, що тип *Stock* можна використовувати для оголошення статичних об'єктів, для створення динамічних об'єктів, для оголошення масивів. Елементи масиву можна ініціалізувати звичайним чином:

```
Stock sally("IBM",10,300.5);
Stock* peter = new Stock("AT&T",25,245);
Stock bella[5] = {
     Stock("NanoSmast", 20, 12.5), Stock("Boffo Objects", 200, 2.0),
     Stock(), Stock("Fleep Enterprises", 60, 6.5)};
```

Елементи масиву *bella* з індексами 0, 1 ,3 ініціалізує конструктор для створення, а з індексами 2, 4 – конструктор за замовчуванням (явно для 2-го і неявно для 4-го).

Перебирати елементи масиву об'єктів також можна звичайним чином: або за допомогою індекса, або з використанням вказівників

```
for (int i=0; i<5; i++) bella[i].show(); // надрукує всі елементи масиву
//         або
for (Stock* tmp=bella, Stock* last=&bella[5]; tmp!=last; tmp++)
    tmp->show();
```

У одній з попередніх лекцій ми говорили про діапазони видимості змінних і функцій: глобальний і локальний. Змінні, оголошені в блоці, мають локальний діапазон видимості, обмежений цим блоком. Змінні, оголошені поза будь-якими функціями, мають глобальний діапазон, а функції завжди мають глобальний діапазон видимості. На додачу до локального та глобального клас визначає власний діапазон видимості – діапазон класу. Всі імена членів класу мають діапазон видимості класу і невидимі зовні (у цьому аспекті клас схожий до простору імен). Ця обставина означає, що, по-перше, різні класи можуть містити одноіменні члени – це не викликає жодних проблем. Всередині класу – у його оголошенні чи визначенні методу – використовують некваліфіковані імена, а клас приховує їх від цілого світу. По-друге, для доступу до членів класу ззовні потрібно якимось чином кваліфікувати їхні імена. Залежно від контексту, це можна зробити за допомогою оператора членства «.» (застосовують до

екземпляра класу), оператора непрямого членства «–>» (застосовують до вказівника на екземпляр) чи оператора визначення діапазону доступу «::» (застосовують до класу).

Ще одна можливість споріднює клас з простором імен – це можливість оголошувати типи (наприклад, перелік, структуру) всередині класу, для внутрішнього, так би мовити, використання. Наприклад, у класі *Stock* було б доцільно розташувати таке оголошення:

```
class Stock
{private:
     enum { Len = 30 };
     char company[Len];
        .....
 public: ..... };
```

У класі оголошено перелік – без імені, з одним значенням. Правду кажучи, це зроблено лише заради цього значення: 30 є «чарівним числом» для назви компанії та всіх методів, що працюють з цією назвою. Відомо, що такі числа варто робити символьними константами. Оголошення переліку вирішило завдання. У наступних проектах ще будуть оголошення типів всередині класу. Наприклад, для створення класу *Queue* буде використано вкладене оголошення структури *Node*.

Ще один спосіб створення всередині класу символьної константи – це оголошення поля даних класу (не екземпляра). У C++ такі члени класу називають статичними.

```
class Stock
{private:
     static const int Len = 30;
     char company[Len];
        .....
 public: ..... };
```

Надалі ми познайомимося ще зі статичними змінними та методами.

Остаточний варіант проекту «маніпулювання пакетами акцій» шукайте на Google Drive. Ви не знайдете у ньому константи *Len*, проте є цікавіші статичні члени (змінні) класу: лічильник створених екземплярів, екземпляр-зразок для конструктора за замовчуванням та методи доступу до них. Виконана початкова ініціалізація статичних змінних. Пропонуємо читачеві самостійно розглянути код тестової програми та пояснити отримані результати.

«Заб'ємо декілька цвяшків» на пам'ять про цей проект: оголошення класу, поділ на приховану та відкриту частини, визначення методів, конструктори (для створення і за замовчуванням), деструктор; використання специфікатора *const*, вказівника *this*, посилання на об'єкт; визначення об'єкта, створення динамічного об'єкта, визначення масиву об'єктів; діапазон видимості класу, вкладені оголошення, статичні члени класу.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою C++.
2. Стивен Прата Язык программирования C++.
3. Бьерн Страуструп Язык программирования C++.