Лекція №1

Тема: Поняття класу та об’єкту класу.

ПЛАН

1. Основи та основні визначення ООП
2. Поняття класу
3. Специфікатори доступу в класах.
4. Функції та члени класу
5. Зв'язок між структурами й класами
6. Зв'язок між об'єднаннями й класами
7. Безіменні об'єднання

1. Основи та основні визначення ООП

Як ми уже відзначали, мова C++ є універсальною. У першій частині курсу розглянуті його імперативні (чи процедурно-орієнтовані) властивості, що, в основному, забезпечуються засобами мови С і деякими додатковими механізмами (посиланнями, inline-функціями, операторами приведення типів і ін.). Основу парадигми імперативного програмування складає алгоритм, що обробляє дані. У рамках цього підходу алгоритм і дані відділені один від одного. Головна роль в імперативному програмуванні приділяється алгоритму, а дані знаходяться в тіні.

Загалом імперативний підхід можна описати в такий спосіб. Програма розбита на модулі, що реалізують різні алгоритми, як правило вузькоспеціалізовані. Модулі одержують дані, обробляють їх і передають іншим модулям. Координація роботи здійснюється головним модулем: він викликає підпрограми, передає їм дані і повертає результат програми. Модулі є відносно самостійними. Вони можуть викликати один одного, самих себе і навіть головний модуль, вводити з зовнішнього світу додаткові дані і виводити результати.

Потреби програмування, поставленого на промислову основу, неможливо повною мірою задовольнити за допомогою імперативного програмування. Навіть усередині однієї команди необхідно погоджувати деталі представлення даних і реалізації алгоритмів, спрямованих на рішення загальної задачі. Програмісти, що не погодили свої дії, ризикують створити несумісні модулі. Незначне відхилення від загального плану може привести до серйозних наслідків. У силу цих причин виникла необхідність створити таку концепцію програмування, що підвищила б продуктивність і надійність роботи програмістів, дозволивши застосувати конвеєрні методи роботи. Так виникла парадигма об'єктно-орієнтованого програмування.

В основу об'єктно-орієнтованого програмування покладене поняття об'єкта і принципи інкапсуляції, приховання інформації, абстракції даних, успадкування і поліморфізму.

Основна ідея об'єктно-орієнтованого програмування полягає в тім, що всі обчислення відбуваються усередині об'єктів, що обмінюються між собою повідомленнями. У деякому сенсі об'єктно-орієнтоване програмування можна вважати вищим ступенем розвитку модульного програмування; тільки роль модулів тут грають об'єкти. В імперативному програмуванні вся інформація оброблялася усередині модулів, причому дані не залежать від них. Тепер об'єкти наділені можливістю не тільки зберігати, але й обробляти дані. Наприклад, якщо в імперативних програмах структура являє собою просту сукупність записів, то в об’єктно-орієнтованих — це активний об'єкт, що не просто зберігає інформацію, але й обробляє її. У цьому і полягає принцип інкапсуляції.

Отже, об′єкт — це об′єднання даних і алгоритмів, що обробляють ці дані. Тепер ми можемо занурити масив і метод його сортування в деякий об'єкт, а потім звернутися до цього об'єкта з проханням упорядкувати елементи масиву і видати результат. Основна перевага такого підходу — гнучкість. Якщо даний модуль є частиною більш складної конструкції, необхідно лише погодити способи його оголошення з зовнішнім світом — те, що називають інтерфейсом. Його внутрішній світ інших програмістів не стосується. Як зберігаються дані усередині модуля і як вони обробляються, повинний знати тільки творець даного об'єкта. А якщо інші програмісти виявляться занадто безцеремонними і захочуть змінити внутрішню структуру об'єкта? Як захистити його від стороннього втручання? Для цього використовується принцип приховання інформації. В об’єктно-орієнтованому програмуванні внутрішню структуру об'єкта можна розділити на три розділи: відкритий, закритий і захищений. Не вдаючись у передчасні подробиці, помітимо, що відкритий розділ доступний усім сутностям програми (інакше кажучи, видний з будь-якої її точки), закритий розділ — лише внутрішнім сутностям об'єкта, а захищений — тільки об'єктам, створеним з даного об'єкта (спадкоємцям).

Для того щоб програма була ефективною, необхідно, щоб об'єкти добре відповідали розв'язуваній задачі. Однак згодом об'єкт може морально застаріти. Наприклад, може з'явитися новий надшвидкісний алгоритм сортування. Що робити? Створити новий об'єкт? Змінити старий? Для того щоб полегшити задачу, вирішили розділити схему об'єкта і його реалізацію. Це дозволяє програмісту описати, що може робити об'єкт, не конкретизуючи, як це робити. У цьому полягає принцип абстракції даних. Принципова схема об'єкта не повинна залежати від її конкретного наповнення. Необхідно лише вказати, що об'єкт повинний містити певну операцію. Якщо алгоритм знадобиться змінити, ми модифікуємо його реалізацію, але загальна структура об'єкта від цього не постраждає.

Уявимо собі, що нам знадобилося створити новий об'єкт, що володів би можливостями старого, але вмів би робити і щось нове. Що робили в минулому? Копіювали зміст старого модуля і додавали в нього нові функції.

Тепер усе це можна робити набагато простіше! Ми просто створюємо об'єкт, що успадковує усі властивості свого попередника, і додаємо в нього нові можливості. Цей механізм називається успадкуванням.

Об'єкт може одержувати різноманітну інформацію. Зрозуміло, для її обробки можна заздалегідь передбачити його реакцію, написавши відповідні функції. Однак в об’єктно-орієнтованих мовах існує можливість перевантажувати

операції і функції, що дозволяє об'єкту самому конкретизувати їхній зміст у ході виконання програми. Що дозволяє спростити програмування. Наприклад, якщо перевантажити оператор +, то операція a + b може мати різний сенс. Якщо ці об'єкти — числа, застосовується звичайна операція додавання, якщо матриці — матрична (конечно, для цього необхідно визначити цю операцію). У цьому (і не тільки) полягає сутність поліморфізму. У принципі поліморфізм виявляється в трьох ситуаціях: при перевантаженні функцій, при перевантаженні операторів і при пізнім зв'язуванні.

Описуючи принципи об'єктно-орієнтованого програмування, ми оперували поняттям об'єкт. Однак об'єкт — це фізична сутність, що виникає при виконанні програми, тобто сукупність комірок пам'яті, що зберігають дані і код. Для того щоб створити об'єкт, необхідна його схема: які дані він містить, які функції обробляють ці дані, як організований доступ до цих даних і функцій, що називаються членами об'єкта. У мові C++ схемою об'єкта називається клас.

2. Поняття класу

Об'єктно-орієнтоване програмування і проектування побудоване на класах. Будь-яку програмну систему, побудовану в об'єктному стилі, можна розглядати як сукупність класів, можливо, об'єднаних в проекти, простори імен, рішення. Класи створюються за допомогою ключового слова **class**. Оголошення класу визначає новий тип, що зв'язує код і дані між собою. Таким чином, клас є логічною абстракцією, а об'єкт - її фізичним втіленням. Іншими словами, об'єкт - це екземпляр класу.

Оголошення класу дуже схоже на оголошення структури. Нижче приводиться повна загальна форма оголошення класу, який не є спадкоємцем ніякого іншого класу.

class *ім'я-класу { закриті дані й функції специфікатор доступу: дані й функції специфікатор доступу: дані й функції //... специфікатор доступу: дані й функції } список об'єктів;*

3. Специфікатори доступу в класах.

Список об'єктів указувати не обов'язково. Він просто дозволяє повідомляти об'єкти класу. У якості специфікатора доступу використовується одне із трьох ключових слів мови C++:

public
private
protected

За замовчуванням функції й дані, оголошені усередині класу, уважаються закритими й доступні лише функціям - членам цього класу. Специфікатор public

відкриває доступ до функцій і даним класу з інших частин програми. Специфікатор доступу protected необхідний тільки при наслідуванні класів. Зона впливу специфікатора доступу простягнеться до наступного специфікатора або до кінця оголошення файлу.

Усередині оголошення класу специфікатори доступу можна чергувати в довільному порядку. Наприклад, деякі члени класу можна зробити відкритими, а потім знову перейти до оголошення закритих змінних і функцій. Проілюструємо сказане наступним прикладом.

```
#include <iostream>
#include <cstring>
using namespace std;
class employee {
       char name[80]; // Закритий за замовчуванням
public:
       void putname(char *n); // Відкриті члени
       void getname(char *n);
private:
       double wage; // Знову оголошуються закриті члени
public:
       void putwage(double w); // Знову оголошуються відкриті члени
       double getwage();
};
void employee::putname(char *n) { strcpy(name, n); }
void employee::getname(char *n) { strcpy(n, name); }
void employee::putwage(double w) { wage = w };
double employee::getwage() { return wage; }
int main()
{
       employee ted;
       char name[80];
       ted.putname("Тед Джонс");
       ted.putwage(75000);
       ted.getname(name);
       cout << name << " заробляє $";
       cout << ted.getwage() << " за рік.";
       return 0;
    }
```

Тут клас **employee** використовується для зберігання імені співробітника й величини його окладу. Зверніть увагу на те, що специфікатор доступу **public** використаний двічі.

Незважаючи на тещо обмежень на використання специфікаторів доступу немає, цією можливістю не варто зловживати. Вона корисна лише для візуалізації різних груп змінних, зв'язаних між собою, і поліпшує наочність програми. Однак компіляторові однаково, як згруповані дані усередині оголошення класу. Багато програмістів уважають більш природнім групувати всі відкриті, захищені й закриті дані, використовуючи не більш одного розділу кожного виду. Наприклад, більшість програмістів записала б оголошення класу employee так, як показано нижче.

```
class employee
{
       char name[80];
       double wage;
public:
       void putname(char *n)
       void getname(char *n)
       void putwage(double w)
```

```
        double getwage()
        };
```

В більшості простих задач, реалізацію методів класу можна прописувати при описі класу. Це виглядатиме наступним чином:

```
class employee
{
        char name[80];
        double wage;
public:
        void putname(char *n)
                { strcpy(name, n); }
        void getname(char *n)
                { strcpy(n, name); }
        void putwage(double w)
                { wage = w; }
        double getwage()
                { return wage; }
};
```

4. Функції та члени класу

Функції, оголошені усередині класу, називаються функціями-членами. Вони мають доступ до всіх елементів свого класу, включаючи закриті члени. Змінні, оголошені усередині класу, називаються змінними- членами або дані-члени. (Використовується також термін екземпляр змінної.) Отже, будь-який елемент класу називається членом класу.

Існує кілька обмежень на застосування членів класу. Нестатичні члени класу не можуть мати ініціалізатор. Оголошення класу не може містити оголошення свого об'єкта. (Хоча членом класу може бути покажчик на об'єкт даного класу.) Члени класу не можуть оголошуватися із ключовими словами auto, extern і register.

Як правило, усі дані-члени оголошують закритими. Це забезпечує інкапсуляцію даних. Однак існують ситуації, у яких деякі змінні слід повідомляти відкритими. (Наприклад, якщо деяка змінна використовується дуже часто, відкритий доступ виконується набагато швидше.) Синтаксична конструкція для доступу до відкритого члена класу нічим не відрізняється від виклику функції-члена: вказується ім'я об'єкта, оператор "." і ім'я змінної. Розглянемо простий приклад.

```
include <iostream>
using namespace std;
class myclass {
public:
        int i, j, k; // Доступні з будь-якої точки програми
};
        int main() {
                myclass a, b;
                a.i = 100; // Змінні i, j так доступні
                a.j = 4;
                a.k = a.i * a.j;
                b.k = 12; // Нагадаємо, що змінні a.k і b.k
                                // належать різним об'єктам
                cout << a.k << " " << b.k;
                return 0;
        }
```

5. Зв'язок між структурами й класами

Структури є спадщиною мови С. Як ми вже знаємо, синтаксичні конструкції класу й структури дуже схожі. Однак зв'язок між структурою й класом набагато тісніше, чим видасться на перший погляд. У мові С++ роль структури була значно розширена.

Фактично вона стала альтернативою класу. Єдина відмінність між структурою й класом полягає в тому, що всі члени структури за замовчуванням уважаються відкритими, а всі члени класу - закритими. В усі інших відносинах структури й класи еквівалентні. Інакше кажучи, у мові C++ структура є різновидом класу. Розглянемо приклад програми, у якій структура використовується як клас, що контролює доступ до рядка.

```
include <iostream>
include <cstring>
using namespace std;
struct mystr {
       void buildstr(char *s); // Відкритий член
       void showstr();
private: // Закритий член
       char str[255];
};
void mystr::buildstr(char *s) {
       if(!*s) *str = '\0'; // Ініціалізація рядка
else strcat(str, s);
}
void mystr::showstr() {
       cout << str << "\n";
}
int main() {
       mystr s;
       s.buildstr(""); // Ініціалізація
       s.buildstr("Усім "),
              s.buildstr("привіт!");
       s.showstr();
       return 0;
    }
```

Ця програма виводить на екран рядок "усім привіт!".

Клас mystr можна переписати за допомогою ключового слова **class**

```
class  mystr {
       char  str[255];
public:
       void buildstr(char  *s);   //   Відкритий член
       void showstr();
    };
```

Виникає природне запитанння: "Навіщо в мові C++ передбачено два еквівалентні ключові слова **struct** і **class**?". Ця уявна надмірність пояснюється декількома причинами. По-перше, не потрібно обмежувати можливості структур. У мові C вони вже використовувалися для групування даних. Отже, можна зробити невеликий крок уперед і включити в структури функції-члени. По-друге, оскільки структури й класи тісно зв'язані між собою, програми, написані мовою C, легко трансформувати в програми мовою C++. І, нарешті, незважаючи на те що структури й класи практично еквівалентні, застосування двох альтернатив зберігає волю модифікації для ключового слова **class**, оскільки ключове слово **struc**t назавжди пов'язане з мовою C и забезпечує зворотну сполучність програм.

Хоча ключове слово **struc**t може заміняти слово **class**, більшість програмістів віддають перевагу цього не робити. Звичайно ключове слово **class** слід застосовувати для оголошення класу, а ключове слово **struct** - для оголошення структури в стилі C. Саме такий стиль прийнятий у нашій книзі. Іноді для опису структуриу стилі C застосовується абревіатура POD (Plain Old Data - прості старі дані). Це значить, що така структура не містить конструкторів, деструктора й функцій- членів.

6. Зв'язок між об'єднаннями й класами

Крім структур, для визначення класу використовуються об'єднання union. У мові C++ об'єднання можуть містити не тільки дані, але й функції. Вони можуть включати конструктори й деструктор. Об'єднання зберігають усі властивості, передбачені мовою C, зокрема , їх члени можуть розміщатися в одній і тій же області пам'яті. Як і структури, члени об'єднання за замовчуванням уважаються відкритими й повністю сумісні з мовою C. У наведеному нижче прикладі об'єднання використовується для перестановки байтів у змінній типу **unsigned short.** (Передбачається, що ціле значення займає 2 байт.)

```
#include <iostream>
using namespace std;
union swap_byte{
       void swap();
void set_byte(unsigned short i);
void show_word();
unsigned short u;
unsigned char з[2]; };
void swap_byte::swap() {
       unsigned char t;
       t = c[0]; c[0] = c[1]; c[1] = t;
}
void  swap_byte::show_word() {
       cout << u;
}
void swap_byte::set_byte(unsigned short i) {
       u = i;
}
int main() {
       swap_byte b;
       b.set_byte(49034);
       b.swap();
       b.show_word();
       return  0;
    }
```

Оголошення об'єднання в мові C++ визначає особливий вид класу. Це значить, що принцип інкапсуляції не порушується.

Існує кілька обмежень, накладених на застосування об'єднань у мові C++. По-перше, об'єднання не можуть використовувати механізм наслідування. По-друге, об'єднання не може служити базовим класом. Об'єднання не може містити: 1) віртуальні функції; 2) статичні змінні й посилання; 3) об'єкти класів, у яких перевантажений оператор присвоювання; і, нарешті, 4) об'єкти класів, у яких явно задані конструктори або деструктор.

До об'єднань, що не містять функції-члени, конструктори й деструктор, також можна застосовувати термін POD.

7. Безіменні об'єднання

У мові C++ є особливий різновид об'єднань - безіменні об'єднання, які не мають типу й не можуть утворювати об'єкти. Замість цього безіменне об'єднання повідомляє компілятора, що його члени зберігаються в одній області пам'яті. Однак доступ до цих змінних здійснюється безпосередньо, без допомоги оператора ".". Розглянемо наступну програму.

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
```

```
    // Визначення безіменного об'єднання
    union{ long l; double d; char s[4]; };
    // Тепер відкритий прямий доступ до елементів об'єднання.
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;
    return 0;
}
```

Як бачимо, елементи об'єднання нічим не відрізняються від звичайних локальних змінних. Незважаючи на те що вони оголошені усередині об'єднання, їх область видимості поширюється на весь блок. Звідси випливає, що імена членів безіменних об'єднань не повинні конфліктувати з іншими ідентифікаторами в тій же області видимості.

Усі обмеження, накладені на звичайні об'єднання, поширюються й на безіменні. По-перше, у них можуть утримуватися лише дані (функції-члени не допускаються). По-друге, безіменні об'єднання не можуть містити закриті й захищені елементи. На закінчення, глобальні безіменні об'єднання повинні оголошуватися за допомогою ключового слова **static**.