



Лекція 3.

Класи та об'єкти. Проектування класів та їх методів.

Мета введення класів в С++

- Головна мета введення концепції класів в С++ — це забезпечення програміста засобами для створення нових типів, які були б такими ж зручними у використанні, як і вбудовані.
- **Тип** — конкретний представник деякої концепції.
- **Клас** — це тип, що визначає користувач. Для визначення концепції, яка не виражається безпосередньо вбудованими типами, створюються нові типи. Ретельно підібраний набір типів, які визначає користувач, робить програму коротшою і виразнішою.
- Основний сенс введення нових типів полягає в обмеженні доступу до даних ззовні та у використанні для цього спеціальних процедур у межах чітко визначеного інтерфейсу.

Функції-члени

- Функції, визначені всередині опису класу (до речі, структура — це один з видів класу), називаються **функціями-членами**, і їх можна викликати тільки для змінної відповідного типу, використовуючи стандартний синтаксис доступу до членів структури.
- Оскільки різні структури можуть мати функції-члени з однаковими назвами, то, визначаючи функцію-член треба вказати ім'я структури:

Приклад

```
class complex_c1 {  
    public://need to know style- our preference  
        void assign(double r, double i);  
        void print()  
            { cout << real << " + " << imag << "i ";}  
    private:  
        double real, imag;  
};  
inline void complex_c1::assign(double r, double i = 0.0)  
{  
    real = r;  
    imag = i;  
}
```

Контроль доступа

Розглянемо приклад:

```
class X
{
public:
    void init();
    int getITnow();
private:
    int x,y,z;
    int px,py,pz;
};
```

Контроль доступу

- Імена в закритій `private` частині можна використовувати тільки у функціях-членах класу. Відкрита `public` частина утворює відкритий інтерфейс об'єктів класу. (Структура — клас, члени якого відкриті за замовчуванням). Крім того, існує мітка `protected` (захищений), тобто всі змінні будуть доступні тільки прямим нащадкам цього класу.
- Захист закритих даних базується на обмеженні використання імен членів класу. Цей захист можна обійти, маніпулюючи з адресами і явним перетворенням типу. Захист проти зловмисного доступу до закритих даних мовою високого рівня можна забезпечити тільки на апаратному рівні, хоча навіть це — досить складне завдання в реальній системі.

Статичні члени

Змінну, яка є частиною класу, а не частиною об'єкта цього класу, називають *статичним членом* і позначають специфікатором **static**

Приклад

```
#include <iostream>
using namespace std;
class Conscription {
    static int Age;
public:
    void setInt(int n) {
        Age = n;
    }
    int getInt() {
        return Age;
    }
};

int main()
{
    Conscription Petrov, Sidorov;

    Petrov.setInt(18);

    cout << "Petrov's age: " << Petrov.getInt() << '\n'; // displays 18
    cout << "Sidorov's
age: " << Sidorov.getInt() << '\n'; // also displays 18

    return 0;
}
```


Константні функції-члени

Нехай у класі X існують функції, які надають і змінюють значення об'єкта типу X . Але, на жаль, не існує способу для перевірки значення об'єкта X . Проте цю проблему можна легко вирішити, описавши ці функції як **константні функції-члени**, тобто функції, які не змінюють стан X :

Приклад: три способи використання const

```
class Birthday {  
    int d,m,y;  
    const string congratulations;  
    public:  
        int day() const {return d;}  
        int month() const {return m;}  
        int year() const {return y;}  
        const string getCon() { return congratulations; }  
    void print (const int d, const int m, const int y)  
        {cout<<"My birthday is "<< d<<"."<<m<<"."<<y<<".";}  
        //...  
};
```

Константні функції-члени

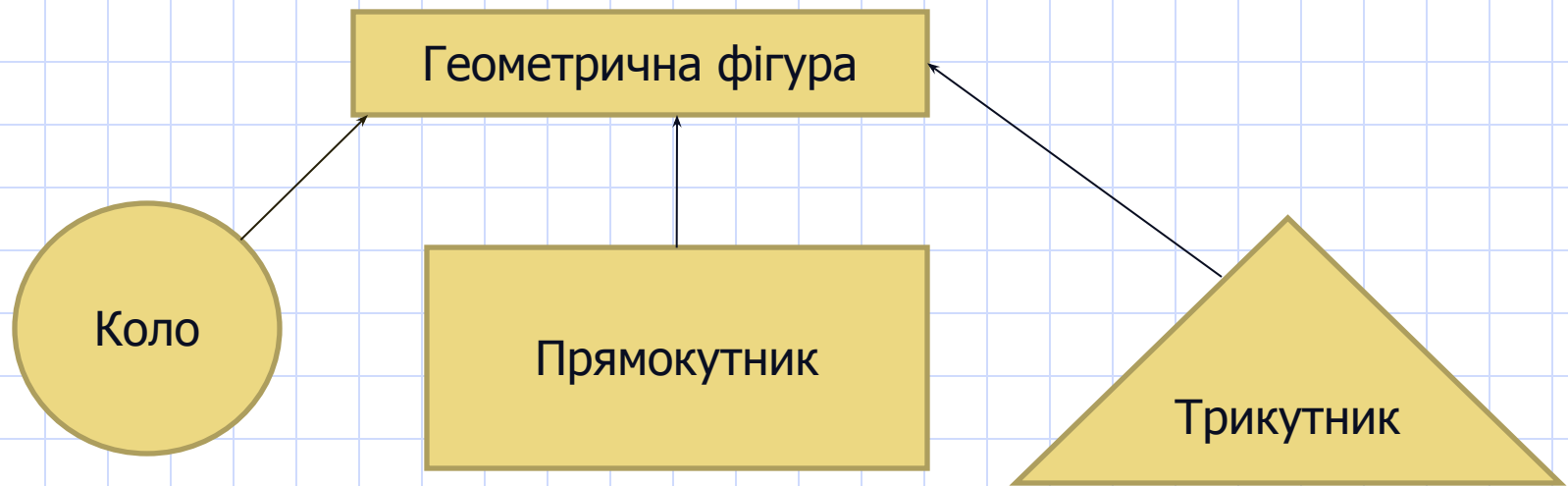
- Коли константна функція-член описується зовні, а не всередині класу, то потрібно додати суфікс `const`:

```
inline int Date::year() const //правильно
{ return y;}
```

- Константну функцію-член можна викликати як для константного, так і для неконстантного об'єкта, тоді як неконстантну функцію-член можна викликати тільки для об'єкта, який не є константою

Підкласи

Підкласи — це класи, які успадковують усі "властивості" суперкласу ("батьківського класу")



Віртуальні функції

Віртуальні функції визначаються специфікатором **virtual** і дозволяють програмісту описати в базовому класі функції, які можна було б замінити у кожному наступному класі.

Ієрархія класів

- Об'єкти різних класів і самі класи можуть перебувати у відношенні успадкування, за якого формується ієрархія об'єктів, що відповідає заздалегідь передбаченій ієрархії класів.
- Ієрархія класів дозволяє визначати нові класи на основі вже існуючих. Існуючі класи зазвичай називають **базовими** (інколи **батьківським**), а нові класи, що формуються на основі базових, — **похідними** (**породженими**), інколи **класами-нащадками** або **спадкоємцями**. Похідні класи “отримують спадок” — дані і методи своїх базових класів — і, крім того, можуть поповнюватись власними компонентами (даними і власними методами).
- Наприклад, за таким визначенням
`class S: X{...};`
клас S породжений класом X, звідки він успадковує компоненти.

Приклад виконання

```
#include <string>
#include <conio.h>
#include <iostream>
```

```
using namespace std;
```

```
class Student
{
private:
    string name;
    int age, course;
public:
    void setData();
    void getData();
};
```

Приклад виконання

```
void Student::setData()
{
    cout<<"Enter name"<< endl;
    cin>>name;
}
void Student::getData()
{
    cout<< " Name=" <<name<<";"<<endl;
    cout<< " Age=" <<age;
}
```


Приклад виконання

```
//-----  
int main()  
{  
    Student Olga; //створення об'єкта  
    Student* Nick; //створення вказівника  
    Olga.setData(); //виклик функції setData() об'єкта Olga  
    Olga.getData();  
    Nick=new Student; //виділення пам'яті під об'єкт  
    Nick->setData(); //виклик функції об'єкта *Nick  
    Nick->getData();  
    getch();  
    return 0;  
}
```

Об'єкти

- *Об'єкт завжди:*
 - є чітко обмеженим
 - не обов'язково є відчутним
- **Приклад:** процес хімічного виробництва
- *Об'єкт характеризується станом, поведінкою та ідентичністю;*
структура та поведінка схожих об'єктів визначають їх загальний клас. Об'єкт - це екземпляр класу.
- **Приклад:** автомат з напоями, ліфт.
- *Стан об'єкту* — перелік (зазвичай статичний) усіх властивостей цього об'єкту та поточними (звичайно динамічними) значеннями кожного з цих властивостей.

Приклад

```
class PersonnelRecord
{
public:
    char * employee Name () const; //всі об'єкти можуть
    int employee Local Security Number () const;
                                //отримати дані
    char * employee Department () const;
protected:
    char name [100];           //лише підкласи
    int Social Security Number; //можуть визначати
    char department [10];     //значення
    float Salary;
};
```

Поведінка об'єкту

- *Поведінка об'єкту* — це спосіб дії та реакції об'єкту.
- Поведінка висловлюється в термінах стану об'єкту та передачі повідомлень.
- *Поведінка* — яка спостерігається і перевіряється зовні дії об'єкту.
- **Приклад:** Автомат — в залежності від стану (кількості монет) видає або ні напій.

Приклад

- **Розглянемо клас «Черга»**

```
class Queue
{
public:
    Queue ();
    Queue (const Queue &);
    virtual ~Queue (); //знищує чергу, але не її учасників
    virtual Queue & Operator = (const Queue &)
        //virtual - буде визначатися
        // в класах-нащадках
    virtual int operator = = (const Queue &) const;
    int operator! = (const Queue &) const;
    virtual void clear ();
    virtual void append (const void *);
    virtual void pop (); //просування черги
    virtual void remove (int at);
    virtual int length ();
    virtual int isEmpty () const;
    virtual const void * front () const;
    virtual int location (const void *)const;
protected:
};
```

Mutable

mutable — антипод `const`, визначає член, який не буде `const` ні за яких умов (навіть для `const` об'єкта).

Відношення між класами

Класи не існують ізольовано.

Основні типи відносин між класами:

1. "узагальнення/спеціалізація" (загальне–частинне) "is a"
2. "ціле–частина" — "part of"
3. асоціація — семантичний, смисловий зв'язок.

ОО мови програмування підтримують різні комбінації наступних типів відносин:

- асоціація — найбільш загальне та невизначене відношення
- успадкування — "загальне–частинне"
- агрегація — "ціле–частина"
- використання — наявність зв'язку між екземплярами класів
- інстанціонування — специфічний різновид узагальнення
- метаклас — клас класів (класи як об'єкти).

Приклад

Приклад

Студент КПІ - студент

Студент мед. Університету - студент

Студенти фіз-теху та ФІОТу - студенти КПІ

Відмінники - складові частини обох типів студентів

Викладачі - наводять жах на студентів.

Асоціація

- **Приклад** — товари та продаж.
- Class Product; //те, що продали
Class Sale; //угода, в якій продано
//декілька товарів

```
Class Product {  
    public:  
        ...  
    protected:  
        Sale* last Sale;  
};  
Class Sale {  
    public:  
        ...  
    protected:  
        Product** product Sold;  
};
```

Асоціація

- *Асоціація* — смисловий зв'язок, як правило, не має напрямку та не пояснює, як класи спілкуються один з одним.
- *Потужність* — кількість учасників цього смислового зв'язку
 - один до одного;
 - один до багатьох;
 - багато до багатьох.
- *Агрегація* — включення одного класу до іншого — відповідає *агрегації* між екземплярами.
- Агрегація як співвідношення "ціле–частина" є *спрямованою*. Не вимагає обов'язкового фізичного включення (акціонер *володіє* акціями, але не складається з них).
- Якщо (і тільки якщо) існує відношення "ціле–частина" між об'єктами, класи повинні знаходитися у співвідношенні *агрегації*.
- *Використання* — спрямовано, якщо клас є частиною сигнатури функції-члена іншого класу (параметром).
- Використання класів → рівноправний зв'язок між їх екземплярами (*клієнт–сервер*).

Конструктори

Одне з основних завдань об'єктно-орієнтованого програмування полягає у тому, щоб об'єкти описаного раз і назавжди класу працювали «правильно» — тобто так, як це визначає модель. Кожний об'єкт перед тим як почати роботу, потрібно створити, тобто перевести в якийсь початковий стан. Отже, треба якимось чином описати можливі механізми створення об'єктів даного класу. Для цього в мові C++ існують **конструктори**. Це особливі методи класу, які й повинні перевести об'єкт у той самий початковий стан. Конструктор описується як метод, ім'я якого збігається з іменем класу, а тип поверненого значення опущений.

Приклад. Конструктор з параметрами для класу «Point»

```
class Point
{
public:
    Point(int x0, int y0);
private:
    int x, y;
};
Point::Point(int x0, int y0)
{
    x=x0;
    y=y0;
}
```

Тепер для створення об'єкта класу Point потрібно після імені змінної вказати параметри, як для виклику функції:

```
Point A(1, 1), B(2, 0);
```

Типи конструкторів

- Існують деякі типи конструкторів, які, крім безпосереднього використання, автоматично викликаються у деяких особливих ситуаціях.
- **Конструктор за замовчуванням**
- Конструктор за замовчуванням - це конструктор, що викликається без параметрів:
Point();
Point(int a=5);
- Його використовують для створення масиву об'єктів, оскільки не зрозуміло, які конструктори і з якими параметрами треба викликати для кожного елементу масиву.
Наприклад:
Point A[10];
Point* B=new Point[10];
- Конструктор за замовчуванням викликається також тоді, якщо не вказано параметри для ініціалізації об'єкта, як у цьому випадку:
Point p;

Конструктор копіювання

- Цей конструктор викликається тоді, коли потрібно створити копію об'єкта. Аргументом цього конструктора має бути посилання на об'єкт цього самого класу:
`Point(Point& p);`
- Важливим випадком, коли викликається конструктор копіювання, є передавання об'єкта у функцію як параметра за значенням. Тоді створюється новий об'єкт і для нього автоматично викликається конструктор копіювання. Створення конструкторів копіювання потрібне у випадку, якщо об'єкт потребує якихось спеціальних операцій при копіюванні, оскільки під час стандартного копіювання вміст одного об'єкта просто побайтно переноситься в інший.

Приклад. Клас String з реалізованими конструкторами

```
class String
{
public:
    String();           // конструктор за замовчуванням
    String(const String& s); // конструктор копіювання
    String(const char* s); // конструктор з параметром
                        // const char*, який являє собою
                        // стандартний рядок s

    ~String();        // деструктор
private:
    char* array;      // масив символів
    int size;         // розмір масиву
};
```

Приклад виклику конструкторів

```
int main()
{ String a, b; // конструктор за замовчуванням
  String c(a); // конструктор копіювання
  print(a);    // конструктор копіювання, оскільки
               // аргумент передається у функцію за значенням
  String d("One"); // конструктор з параметром
//...
}
```


Деструктори

Конструктори ініціалізують об'єкт, тобто вони створюють середовище, у якому "працюють" функції-члени. Іноді створення такого середовища зумовлює "захоплення" якихось ресурсів: пам'яті, файлу, процесорного часу, які повинні бути "звільнені" після їх використання. Тобто класам потрібна функція, яка б знищувала об'єкт аналогічно тому, як його створює конструктор. Такі функції називають *деструкторами*

Приклад деструктора

```
class Column
{
    const char* s;
    // ...
};

class Table
{
    Column* p;
    size_t sz;
public:
    Table(size_t s=15) {p=new Column[sz=s];}
    //конструктор
    ~Table() {delete[] p;}           //деструктор
    // ...
};
```