# Програмування. Лекция 3.

План

1) Дружны функції, класи
2) Перевантаження операторів, операцій
3) Статические переменные
4) Статические методы класса
5)

**(Слайд 7) Перевантаження операторів (операцій)**

**Перевантаження операторів (operator overloading)** - це можливість застосовувати вбудовані оператори мови до різних типів, в тому числі і призначених користувачем. Насправді, це досить стара ідея. Уже в перших мовах програмування символи арифметичних операцій: +, -, etc. використовувалися для операцій над цілими і дійсними числами, незважаючи на те, що вони мають різний розмір і різний внутрішнє подання і, відповідно, ці операції реалізовані по різному. З появою об'єктно-орієнтованих мов ця ідея отримала подальший розвиток.

Якщо операції над типами, які вводить користувач, мають ***подібну семантику з операціями*** над вбудованими типами, то чому б не використати синтаксис вбудованих операторів. Це може підвищити читання коду, зробити його більш лаконічним і виразним, спростити написання узагальненого коду. У C++ перевантаження операторів має серйозну підтримку і активно використовується в стандартній бібліотеці.

**Переваги цього: Перевантаження операцій надає можливість** використовувати власні типи даних як стандартні, а складний та малозрозумілий текст програми перетворювати на інтуїтивно зрозумілий. Позначення власних операцій вводити не можна.

Перевантаження операцій здійснюється за допомогою методів спеціальної форми (функцій-операцій) і підпорядковується таким правилам:

- при перевантаженні операцій зберігаються кількість аргументів, пріоритети операцій та правила асоціації (зліва направо чи справа наліво), які використовуються у стандартних типах даних);
- для стандартних типів даних перевизначати операції не можна;
- функції-операції не можуть мати аргументів за замовчуванням;
- функції-операції успадковуються (за винятком “=”);
- функції-операції не можуть визначатися як static.

**(Слайд 4)**

Функція-операція містить ключове слово ***operator,*** за яким слідує знак операції, яку треба перевизначити:

**<тип> operator <операція> (<список параметрів>)**
**{ <тіло функції> }**

Такий синтаксис повідомляє компіляторові про те, що, якщо операнд належить до визначеного користувачем класу, треба викликати функцію з таким ім’ям, коли зустрічається в тексті програми ця операція.

Функцію-операцію можна визначити трьома способами:

- ✓ як метод класу;

- ✓ як «дружню» функцію класу;
- ✓ як звичайну функцію.

У двох останніх випадках функція повинна мати хоча б один аргумент, який має тип класу, покажчик чи посилання на клас.

**(Слайд 5)**

С++ дозволяє перевантажувати більшість операцій в такий спосіб, щоб стандартні операції можна було використовувати і для об'єктів створюваних користувачем класів.

**Можна перевантажувати будь-які операції, існуючі в С++, за винятком:**

**. .* ? : :: # ## sizeof**

Для согласованности при определении перегруженных операторов рекомендуется следовать модели для встроенных типов. Если семантика перегруженного оператора существенно отличается от его значения в других контекстах, это может скорее запутывать ситуацию, чем приносить пользу

| **Операція** | **Рекомендована форма перевантаження** |
|---|---|
| Всі унарні операції | Метод класу |
| = [] () -> | Метод класу |
| += -= *= /= %= &= ^= | Метод класу завжди |
| Інші бінарні операції | Зовнішня функція friend / метод класу |
| << >> | Тільки зовнішня функція friend |

**(Слайд 6)**

**Перевантаження операторів з використанням методів класу.**

Можливо перевантажувати як бінарні оператори (наприклад, «+», «=»), так і унарні (наприклад, «++», «--»).

При перевантаженні унарної операції функція-оператор не має параметрів, тобто жоден об'єкт не передається явним чином, а операція виконується над об'єктом, який генерує виклик цього методу через неявно переданий покажчик *this*.

**(Слайд 7)**

Розглянемо перевантаження унарних операцій на прикладі операція **інкременту (префіксна форма)**.

```
class Point
{
   int x,y,z;
public:
   Point operator++();
};
  Point Point::operator++()
  {
     ++x; ++y; ++z; //інкремент координат x, y, z
return *this; //повертання значення
  }
…….
Point  a(1,2,3);
++a;
```

**(Слайд 8)** Операція **інкременту (постфіксна форма)**.

```
class Point
{
 int x,y,z;
public:
 Point operator++(int notused);//вказівка компілятору на постфіксну форму, параметр не використовується
};
 Point Point::operator++(int notused)
 {Point temp = *this;//збереження вихідного значення
     ++x; //інкремент координат x, y, z
     ++y; ++z;
  return temp; //повертання вихідного значення
  }
…….
Point  a(1,2,3);
a++;
```

**(Слайд 9)** Перевантаження **<u>бінарних операцій</u>**

- В цьому випадку функція-оператор має тільки один параметр.
- Інший параметр передається неявним чином за допомогою покажчика ***this*** на об'єкт, для якого викликається функція-оператор.
- Об'єкт, що знаходиться справа від знаку операції, передається методу як параметр

**(Слайд 10)** Розглянемо на прикладі бінарної операції +

```
class Point
{  int x,y,z;
public:
   Point operator+ (Point op2);
};
Point Point::operator+(Point op2)
{
   Point temp;
   temp.x=x+op2.x;
   temp.y=y+op2.y;
   temp.z=z+op2.z;
   return temp;
}
……
Point a(1,2,3), b(2,3,4),c;
```

c=a+b;… x,y,z відповідають this->x, this->y, this->z для об'єкту,  що викликає операцію (який зліва)

**(Слайд 11)**

**(Слайд 12) Перевантаження операторів як «дружні» функції**

- «дружні» функції мають бути оголошені в класі;
- «дружні» функції мають доступ до приватних елементів класу;
- «дружні» функції не є членами класу, вони не можуть мати неявний аргумент *this*, тому:
  - при перевантаженні ***бінарних*** операторів обидва операнди передаються функції-оператору,
  - при перевантаженні ***унарних*** операторів передається один операнд.

**(Слайд 13)** Перевантаження унарних операцій на прикладі операції декременту

Префіксна форма

```
friend Point operator--(Point &op1)
{op1.x--;
 op1.y--;
 op1.z--;
 return op1;
}
```

Постфіксна форма

```
friend Point operator--(Point &op1,int noused)
{
Point temp = op1;
op1.x--;
op1.y--;
op1.z--;
return temp;
}
```

**(Слайд 14)** Перевантаження **бінарних операцій** на прикладі операції

Перевантаження **бінарних операцій** на прикладі операції ***

```
class Point {
   int x, y, z;
  public:
    friend Point operator*(Point op1,Point op2);
...};
...
Point operator*(Point op1, Point op2)
  {
    Point temp;
    temp.x = op1.x * op2.x;
    temp.y = op1.y * op2.y;
    temp.z = op1.z * op2.z;
    return temp;
  }…
Point a(1,2,3), b(2,3,4),c;
c=a*b;…
```

**(Слайд 15) Перевантаження операторів з використанням звичайних функцій**

- Звичайні функції не мають доступу до приватних елементів класу, тому доступ до закритих членів класу відбувається через спеціальні методи класу (геттери);
- Механізм перевантаження:
  - при перевантаженні ***бінарних*** операторів обидва операнди передаються функції-оператору,
  - при перевантаженні ***унарних*** операторів передається один операнд.

**(Слайд 16)** Перевантаження **бінарних операцій** на прикладі операції **+**

```
class Point
{
   int x,y,z;
```

```
public:
   int getX(){return x;}
   int gexY(){return y;}
   int getZ() {return z;}
};
three_d operator+ (Point &op1,Point &op2)
{
return Point(op1.getX()+op2.getX(),op1.gexY()+op2.gexY(),op1.getZ()+op2.getZ());
}
```

**(Слайд 17)**Перевантаження **бінарних операцій** на прикладі операції <

```
bool operator <(const Person &M1, const Person &M2)
{
        if( M1.getAge() < M2.getAge())
                return true;
        return false;
}
```

**(Слайд 18) Перевантаження оператору присвоювання =**

- Виконується за допомогою методу класу.
- Операторна-функція повинна повертати посилання на об'єкт, для якого вона викликана, і приймати в якості параметра єдиний аргумент - посилання на об‘єкт, який присвоюється Зазвичай механізм первантаження такий саме і як для перевантаження бін рних операторів методами класу, але є один момент, на який треба звернути увагу.

!!!При спробі присвоїти об’єкт самому собі, спочатку програма видалить дані, у тому числі посилання на пам'ять, а потім спробує це присвоїти. Тому треба робити перевірку на само присвоювання

Обратите внимание, нет необходимости выполнять проверку на самоприсваивание в конструкторе копирования. Это связано с тем, что конструктор копирования вызывается только при создании новых объектов, а способа присвоить только что созданный объект самому себе, чтобы вызвать конструктор копирования — нет.

```
Person& operator = (const Person &M){
        // !Треба робити перевірку на самоприсвоювання:
        if (&M == this) return *this;
    if (name) delete [] name;
        if (M.name){name = new char [strlen(M.name) + 1];
                strcpy(name, M.name);}
        else name = 0;
        age = M.age;
 return *this;}
```

На відміну від інших операторів, компілятор автоматично надасть відкритий оператор присвоювання за замовчуванням для вашого класу при його використанні, якщо ви не надасте його самостійно. В операторі присвоювання за замовчуванням виконується почленное присвоювання (яке є аналогічним почленного ініціалізації, використовуваної в конструкторах копіювання, що надаються мовою C ++ за замовчуванням). Як і з іншими конструкторами і операторами, ви можете заборонити виконання операції присвоювання з об'єктами ваших класів, зробивши оператор присвоювання закритим або використовуючи **ключове слово delete:**

```
Person& operator = (const Person &M)=delete;
```

**(Слайд 19) Перевантаження операції індексування**

**Операція індексування []** перевантажується для того, щоб використовувати стандартний запис C++ для доступу до елементів членів класу. Операція "[]" перевантажується як бінарна операція [9,15,16]. Її можна перевантажувати **тільки для класу і тільки за допомогою методів класу**.
Однак оскільки ця операція зазвичай використовується ліворуч знака “=”, перевантажена функція має повертати власне значення за посиланням.
В наступному прикладі для класу Vector за допомогою перевантаження операції індексування повертається і-тий елемент масиву цілих чисел beg

```
int Vector::operator [](int i)
{
   if(i<0) cout<<"index <0";
   if(i>=size) cout<<"index>size";
   return beg[i];
}
```

Параметр операторної функції **operator[]()** може мати будь який тип даних: ***символ, дійсне число, строка.***
Для того,щоб операція працювала і зліва і справа від оператора присвоювання (тобто якщо бажаємо повернути і-й елемент масиву та присвоїти йому значення( треба у якості повертає мого параметра брати ***посилання*** на тип елемента)

```
int & Vector::operator [](int i);
```

**(Слайд 20). Перевантаження оператора ( )**
C++ дозволяє перевантажувати **оператор виклику функції ()**. При його перевантаженні створюється не новий засіб виклику функції, а операторна функція, якій можна передати довільне число параметрів [16,17]. **Як метод класу**
У загальному випадку при перевантаженні оператора ( ) визначаються параметри, які необхідно передати функції operator()(). А аргументи, які задаються при використанні оператора ( ) в програмі, копіюються у ці параметри. Об’єкт, який генерує виклик операторної функції, адресується покажчиком this. Наприклад, для класу Vector:

```
void Vector:: operator()(int n)
{
   for(int i=0; i<size; i++)
      beg[i]=n*beg[i];
}
```

Або повернути значення елементу двовимірного масиву.

```
…double& Matrix::operator()(int row, int col)
{
   assert(col >= 0 && col < 5);
   assert(row >= 0 && row < 5);
   return data[row][col];
}…
int main()
{
   Matrix matrix;
   matrix(2, 3) = 3.6;
   std::cout << matrix(2, 3);
   return 0;
```

}

**(Слайд 21). Перевантаження операторів вводу >> та виведення <<**

У мові С++ передбачено засіб **вводу** і **виведення** стандартних типів даних, з використанням операторів помістити в потік >> і взяти з потоку <<. Ці оператори вже перевантажені в бібліотеці <iosream> для роботи з різними стандартними типами даних. Включаючи строки та адреси. Але ці оператори можна також перевантажувати для вводу та виведення типів даних, які визначені користувачем [11,16,17].

Функції перевантаження операторів помістити в потік>> та взяти з потоку << не можуть бути членами класу. Для того, щоб вони мали доступ до елементів класу її перевантажують **як дружні функції.**

```
ostream &operator<<(ostream &output, const Vector &v)
{
   if(v.size==0) out<<"Empty\n";
   else
   {
      for (int i=0; i<v.size; i++)
         output<<v.beg[i]<<" ";
      out<<endl;
   }
   return output;
}
istream &operator >>(istream &input, Vector &v)
{
   for(int i=0; i<v.size; i++)
   {
      cout<<">";
      input>>v.beg[i];
   }
   return input;
}
```

**(Слайд 22)**

Зверніть увагу, що згідно об’явленню ці функції повертають посилання на об’єкт типу ostream чи istream. Це дозволяє об’єднати в одному складовому виражені декілька операторів виведення. Операторні функції у цьому випадку мають два параметри. Перший являє собою посилання на потік, якій використовується у лівій частині оператору. Другий являє об’єкт, який стоїть у правій частині оператору. За необхідністю другий параметр також може мати посилання на об’єкт. Саме тіло функції, для розглянутого прикладу, складається з інструкцій виведення чи вводу масиву класу Vector.

```
#include <iostream>
class Point
{
private:
   double m_x, m_y, m_z;
public:
   Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
   {
```

```
    }
    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};
std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Поскольку operator<< является другом класса Point, то мы имеем прямой доступ к членам Point
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";
    return out;
}
int main()
{
    Point point1(5.0, 6.0, 7.0);
    std::cout << point1;
    return 0;
}
```

### 6.1.4. Перевантаження операторів new і delete

Оператори **new** і **delete** також можуть бути перевантажені у мові С++. Для різного роду задач може виникнути потреба створення власної версії цих операторів [5,16].

Для перевантаження використовується наступний формат:

```
// Виділення пам’яті для об’єкту.
void *operator new(size_t size)
 {
//Конструктор викликається автоматично.
  return pointer_to_memory;
 }
// Видалення об’єкта.
void operator delete(void *p)
 {//Звільнення па’мяті. Деструктор викликається автоматично
 }
```

Тип **size_t** спеціально визначено, щоб забезпечити збереження розміру максимально потрібної області пам’яті, яка може бути виділена для об’єкту. Тип **size_t,** це цілочисельний тип без знаку. Параметр **size_t** визначає кількість байтів пам’яті, які потрібні для збереження об’єкту.

```
// Виділення пам’яті для масиву об’єктів.
void *operator new[](size_t size)
 {// Кожний конструктор викликається автоматично
   return pointer_to_memory;
 }
// Видалення масиву об’єктів.
void operator delete[](void *p)
 {//Деструктор для кожного об’єкту масиву викликається автоматично
 }
```

Необхідно також організувати перевірку виділення динамічної пам’яті.

size_t. Базовый беззнаковый целочисленный тип языка Си/Си++. Является типом результата, возвращаемого оператором sizeof. Размер типа выбирается таким образом, чтобы в него можно было записать максимальный размер теоретически возможного массива любого типа. На 32-битной системе size_t будет занимать 32-бита, на 64-битной - 64-бита. Другими словами в тип size_t может быть безопасно помещен указатель (исключение составляют указатели на функции классов, но это

особый случай). Тип size_t обычно применяется для счетчиков циклов, индексации массивов, хранения размеров, адресной арифметики. Хотя в size_t можно помещать указатель, для этих целей лучше подходит другой беззнаковый целочисленный тип uintptr_t, само название которого отражает эту возможность. В ряде случаев использование типа size_t безопаснее и эффективнее, чем использование более привычного программисту типа unsigned.

size_t — это базовый беззнаковый целочисленный memsize-тип, определённый в стандартной библиотеке языков C/C++. Данный тип описан в заголовочном файле stddef.h для языка C и в файле cstddef для языка C++. Типы, определяемые заголовочным файлом stddef.h, расположены в глобальном пространстве, cstddef размещает тип size_t в пространстве имён std. В связи с тем, что стандартный заголовочный файл stddef.h языка C для обеспечения совместимости включается в C++ программы, в этих программах возможно обращение к типу как в глобальном пространстве имён (::size_t, size_t), так и в пространстве имён std (std::size_t)

Важно!Для создаваемых вами классов всегда имеет смысл экспериментировать сперегрузкой операторов. Как показывают примеры этой главы, механизм перегрузкиоператоров можно использовать для добавления новых типов данных в средупрограммирования. Это одно из самых мощных средств C++.

**(Слайд 23)**

**Глобальний доступ до полів класів в C++. Глобальні змінні.**

*Глобальні змінні* — змінні, які було створено поза межами будь-якого блоку коду (*глобальна* або *файлова* область видимості), вони є доступними з будь-якого місця програми і зберігаються в пам'яті до завершення роботи програми. Зазвичай, глобальні змінні об'являють після блоку **#include**, але вище будь-якого іншого коду, наприклад [3,14,18]:

На відміну від глобальних, локальні змінні об'являються всередині блоку, обмеженого фігурними дужками і є видимі тільки всередині цього блоку [19]. Локальна змінна в деякому блоці коду завжди перекриває глобальну, яка має таке ж ім'я. Для того, щоб примусово вказати, що в даному місті блоку має бути використана глобальна змінна використовується оператор вирішення контексту "::":

**Забезпечення глобального доступу є і перевагою і недоліком** глобальних змінних, так як їх використання може значно зменшити об'єм коду програми, але, в той же час, будь-які функції можуть змінювати значення глобальних змінних, тому у складних об'ємних проєктах слід уникати використання таких змінних, тому що це може призвести до непрогнозованих змін значень таких змінних.

**Статичні змінні**

Компромісним рішенням для забезпечення глобального доступу до змінних є використання *статичних змінних*, які також доступні глобально, але кожна така змінна наявна лише в одному екземплярі в пам'яті. Статичні змінні схожі до глобальних за механізмом розміщення в пам'яті, але на статичні додатково діють специфікатори доступу **public** та **private**.

*В C++ есть возможность доступа всех созданных объектов конкретного класса к одной переменной (полю), содержимое которой хранится в одном месте. Для этого объявляют переменную:*

*Такой класс памяти (статический) может использоваться не только для объявления статических полей (переменных класса), но и для методов класса. Память для этого резервируется при запуске*

*программы до явного создания объекта. Поэтому он является как бы единым для всех копий полей класса. Доступ для такой переменной (::) возможен только после инициализации*
Статическим полям можно задать начальные значения один (и только один) раз в области действия файла. Доступ к открытым статическим элементам класса возможен через любой объект класса или посредством имени класса с помощью бинарной операции разрешения области действия.

Статичні поля найчастіше використовуються в таких ситуаціях:

- необхідність контролю загальної кількості об'єктів класу;
- створення єдиної глобальної змінної, до якої мають доступ усі об'єкти класу.

Статичні поля задаються ключовим словом **static**, яке може бути використано як для атрибутів, так і для методів класу. Особливістю елементів, до яких додане ключове слово **static** є те, що вони належать класу, а не об'єкту цього класу, тому можуть бути використані навіть без створення об'єкту класу, і незалежно від кількості створених об'єктів даного класу, в пам'яті буде знаходитись лише одна копія елементу, що об'явлено як статичний.

Таким чином, в пам'яті буде знаходитись завжди лише по одній копії кожної статичної змінної, а кількість копій нестатичних полів буде дорівнювати загальній кількості об'єктів класу.
Статическим полям можно задать начальные значения один (и только один) раз в области действия файла. Доступ к открытым статическим элементам класса возможен через любой объект класса или посредством имени класса с помощью бинарной операции разрешения области действия.

Ключове слово **static** вказується перед вказанням типу даних або методів:
*static тип_змінної ім'яАтрибуту*;
*static тип_значення_що_повертається ім'яМетоду*(...);
Доступ до статичних змінних або функцій відбувається з використанням імені класу та оператору розширення видимості "::":
*Ім'яКласу* :: *ім'яАтрибуту*;
*Ім'яКласу* :: *ім'яМетоду(...)*;
Статичні атрибути класу об'являються в об'явленні класу (або в заголовковому файлі, якщо він є), а ініціалізуються поза блоком-об'явленням в глобальній області видимості. ***Не можна*** ініціалізувати статичні змінні в тілі класу та в його методах. Такий спосіб ініціалізації потрібний для того, щоб уникнути повторної ініціалізації статичної змінної.
Розглянемо приклад використання статичної змінної для підрахунку загальної кількості створених об'єктів класу [20] (файл «main.cpp»):

```
#include <iostream>
using namespace std;
class X // Об'явлення класу
{
        static int n; //Змінна-лічильник створених екземплярів
        static char ClassName[30];
public:
        static int getN() { return n; }
        static char* getClass() { return ClassName; }
        X() { n++; } // конструктор
};
// Використання класу
int X::n = 0; // Ініціалізація приватного атрибуту поза тілом класу через оператор
```

```
// вирішення контексту
char X::ClassName[] = "My Class";

int main()
{
        X a, b, c; // Об’являємо 3 об’єкти класу X
        cout << X::getN() << " objects of Class \"" << X::getClass() << "\""  << endl;
// Звертання до методів класу також через оператор “::”
//cout << X::n <<  endl; помилка, спроба доступу до приватного члену класу
        return 0;
}
```

**Статичні методи**

- Якщо статична змінні об’явлена у розділі **private** до неї має доступ лише відкритий **статичний метод (static).**
  **static int Point::getCount ();**
  До статичної змінної може мати доступ нестатичний метод класу, але його можна викликати лише тільки через об’єкт класу, бо нестатичні методи прив’язані до об’єктів класу, а статичні існують самі по собі
- ***Статичний метод не має покажчика this***, тому що статичні поля і статичні методи існують незалежно від будь-яких об'єктів класу, тобто до них не прив’язані
- Метод класу може бути оголошений як static, якщо він **не має доступ до нестатичних елементів класу.**
- Статичний метод викликається з додаванням перед його ім'ям **імені класу** і бінарної операції оператору розширення видимості **“::”** **Point::getCounter();**
  або через об’єкт класу **p1.getCounter();**
- Статичні поля класу створюються ***в єдиному екземплярі*** незалежно від кількості визначених в програмі об'єктів.
- Всі об'єкти (навіть створені динамічно) поділяють єдину копію статичних полів.

Статические методы могут использоваться для работы со статическими переменными-членами класса. Для работы с ними не требуется создавать объекты класса.

Классы могут быть «чисто статические» (со всеми статическими переменными-членами и статическими методами). Однако, такие классы, по сути, эквивалентны объявлению функций и переменных в глобальной области видимости, и этого следует избегать, если у вас нет на это веских причин.

```
class Account
{ private:
    double sum;
    static int rate;
    const static rate_default=5;//статична константа
public:
    Account(double sum);
    double getIncome();
    static int getRate();
    static void setRate(int r);

};
    int Account::rate = 8;
```

```
    Account::Account(double sum)
    {
        this->sum = sum;
    }
    double Account::getIncome()
    {
        return sum + sum * rate / 100;
    }

//реалізація статичних методів другий раз static не вказується
    int Account::getRate ()   {
        return rate;
    }
    void Account::setRate(int r)
    {
        rate = r;
    }

int main()
{
    Account account1(20000);
    Account account2(50000);
    Account::setRate(5);        // переустанавливаем значение rate
    std::cout << "Rate: " << Account::getRate() << std::endl;
    std::cout << "Rate: " << account1.getRate() << " Income: " << account1.getIncome()
<< std::endl;
    std::cout << "Rate: " << account2.getRate() << " Income: " << account2.getIncome()
<< std::endl;
    return 0;
}
```

Также нередко в классах используют статические константы. Например, сделаем в классе Account переменную rate константой:

```
class Account
{
public:
    const static int rate = 8;
    Account(double sum)
    {
        this->sum = sum;
    }
    double getIncome()
    {
        return sum + sum * rate / 100;
    }
private:
    double sum;
};
int main()
{
    Account account1(20000);
    Account account2(50000);
    std::cout << "Rate: " << account1.rate << "\tIncome: " << account1.getIncome() <<
std::endl;
    std::cout << "Rate: " << account2.rate << "\tIncome: " << account2.getIncome() <<
std::endl;
    return 0;
}
```