Лекція 6 Перевантаженнѐ операторів

У С++ оператори є функціями, в яких значення операндів є параметрами функцій. Як відомо, функції в С++ можна перевантажувати, отже, оператори, як і інші функції, можна перевантажувати для типів даних, які визначаються програмістом. Принциповий виграш від перевантаження операторів полягає в тому, що воно дозволяє органічно інтегрувати нові типи даних в середовище програмування. По суті, перевантаження операторів є одним із проявів поліморфізму, коли програміст використовує один, звичний йому інтерфейс (оператор) для виконання різних операцій з новими типами даних.

Щоб перевантажити оператор, необхідно визначити значення нової операції для класу, до якого вона буде застосовуватися. Для цього створюється функція operator (операційна функція), яка визначає дію цього оператора. Загальний формат функції operator такий:

тип operator # (список аргументів)

{

операція_над_класом

}

Оператори перевантажуються за допомогою функції *operator*.

Тут оператор, що перевантажується, позначається символом "#", а елемент тип являє собою тип значення, що повертається заданої операцією.

Функція-оператор може бути членом класу чи не бути їм. Операторні функції, які не є членами класу, часто визначаються як його «друзі». Операторні функції-члени і функції, які не є членами класу, розрізняються за формою перевантаження. Кожен з варіантів ми розглянемо окремо. Варто відразу зазначити, що є функції-оператори, які можуть бути визначені лише як члени класу і є функції-оператори, які можуть бути визначені лише поза класом.

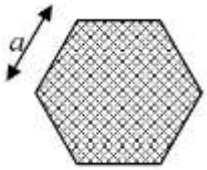
Перевантажити можна тільки оператори, які вже визначені в С++. Створити нові оператори не можна.

Перевантаження операторів з використанням функцій, які не є членами класу

Якщо функція оператора визначена як окрема функція і не є членом класу, кількість параметрів такої функції збігається з кількістю операндів оператора.

Почнемо з простого прикладу. У наступній програмі створюється клас Honeycomb (бджолина сота), який визначає правильний шестикутник зі стороною a.

$$S = \frac{3\sqrt{3}}{2} a^2$$



Для класу Honeycomb перевантажується оператор +. Уявимо собі, що дві бджоли вирішили об'єднати своє майно, для чого їм потрібно замість двох різних сот отримати одну з площею, що дорівнює сумі двох сот. Для цього потрібно обчислити сторону цієї фігури.

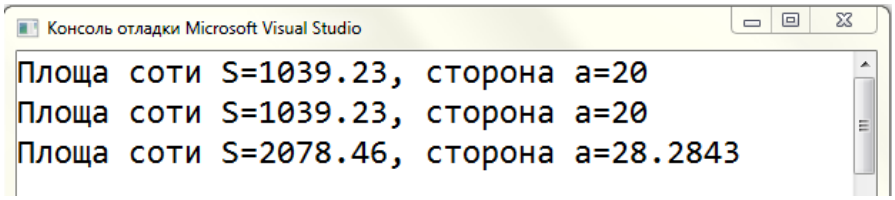
Отже, розглянемо уважно код цієї програми.

```
class Honeycomb
{
 double a, area;
public:
 Honeycomb(double a = 0)
 {
       this->a = a;
       area = 3 * sqrt(3)*a*a / 2;
 }
 // фунуція-оператор + не є членом класу, це дружня функція
 friend Honeycomb operator + (Honeycomb a, Honeycomb b);
 void Show()
 {
       cout << "Площа соти S=" << area << ", сторона a=" << a << endl;
 }
};
```

```
Honeycomb operator + (Honeycomb a, Honeycomb b)
{
 double S = a.area + b.area;
 // Знаходимо сторону нової фігури
 double side = sqrt(2 * S / (3 * sqrt(3)));
 Honeycomb NewHoney(side);
 return NewHoney;
}

int main()
{
 setlocale(0, "Ukr");
 Honeycomb h1(20);
 Honeycomb h2(20);
 Honeycomb hsum=h1 + h2;
 h1.Show(); h2.Show(); hsum.Show();
}
```

При виконанні ця програма генерує такі результати:



При виклику операторної функції +, визначеної поза класом, їй передаються два параметри: перший – той, що стоїть ліворуч, а другий - той, що стоїть праворуч.

З використанням функцій-не членів класу не можна перевантажувати такі оператори:

`=, (), [] , ->.`

Зверніть увагу на те, що функція operator + () повертає об'єкт типу Honeycomb. Незважаючи на те що вона могла б повертати значення будь-якого

допустимого в C++ типу, той факт, що вона повертає об'єкт типу Honeycomb, дозволяє використовувати оператор "+" в таких складових виразах, як a + b + c. Частина цього виразу, a + b, генерує результат типу Honeycomb, який потім складається з об'єктом c. І якби ця частина виразу генерувала значення іншого типу (а не типу Honeycomb), такий складний вираз просто не працював би.

4.2 Перевантаження операторів з використанням функцій-членів класу

При перевантаженні бінарного оператора з використанням функції-члена їй передається явно тільки один аргумент. Другий же неявно передається через this. Таким чином, в рядку

temp.x = x + op2.x;

Під членом x мається на увазі член this-> x, тобто член x зв'язується з об'єктом, який викликає дану операторну функцію. У всіх випадках неявно передається об'єкт, що вказується зліва від символу операції, який став причиною виклику операторної функції. Об'єкт, що розташовується праворуч від символу операції, передається цій функції як аргумент. У загальному випадку при використанні функції-члена для перевантаження унарного оператора параметри не використовуються взагалі, а для перевантаження бінарного – тільки один параметр. Тернарний оператор «?» перевантажувати не можна.

На відміну від оператора "+", оператор присвоювання призводить до модифікації одного зі своїх аргументів. (Перш за все, це становить саму суть присвоювання.) Оскільки функція operator = () викликається об'єктом, який розташований зліва від символу присвоювання (=), саме цей об'єкт і модифікується в результаті операції присвоювання. Після виконання цієї операції значення, що повертається перевантаженим оператором, містить об'єкт, який розташований зліва від символу присвоювання.

Необхідно, щоб функція operator = () повертала об'єкт, що адресується вказівником this, і щоб цей об'єкт був розташований зліва від оператора "=". Це дозволить виконати будь-який ланцюжок присвоювання. Наприклад, щоб можна було виконувати інструкції, подібні наступної:

a = b = c = d;

Операція присвоювання – це одне з найважливіших застосувань вказівника this.

Якщо оператор = не визначений, при виконанні операції присвоювання:

obj1 = obj2;

автоматично створюється побітова копія об’єкта obj2. Якщо в класі не застосовується динамічна пам’ять, яка звільняється деструктором, тоді можна оператор присвоювання не перевантажувати. Однак якщо конструктор виділяє, а деструктор звільняє динамічну пам'ять, необхідно перевантажувати оператор присвоювання так, щоб він виконував дії, подібні до тих, що виконує конструктор копії.

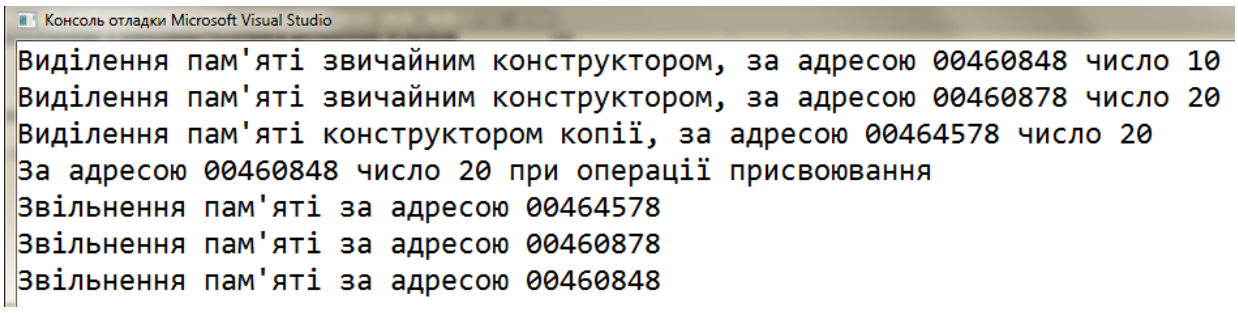
Розглянемо приклад:

```
class MyCl
{
 int *p;
public:
 // звичайний конструктор.
 MyCl(int i=0)
 {
       p = new int;
       *p = i;
       cout << "Виділення пам'яті звичайним конструктором,";
       cout << " за адресою " << p<<" число "<< *p << endl;
 }
 // конструктор копії
 MyCl(const MyCl &obj)
 {
       p = new int;
       *p = *obj.p; // значення копії
       cout << "Виділення пам'яті конструктором копії,";
       cout << " за адресою " << p << " число " << *p << endl;
```

```
 }
 ~MyCl()
 {
       cout << "Звільнення пам'яті за адресою " << p << endl;
       delete p;
 }
 // перевантаження оператора =
 MyCl & operator =(const MyCl &obj)
 {
       *p = *obj.p; // значення копії
       cout << "За адресою " << p << " число " << *p;
       cout << " при операції присвоювання" << endl;
       return *this;
 }
};

int main()
{
 setlocale(0, "Ukr");
 MyCl a(10); // Викликається звичайний конструктор.
 MyCl b(20); // Викликається звичайний конструктор.
 MyCl c = b; // Викликається  конструктор копії.
 a = b; // Викликається  перевантаженний оператор =
}
```

При виконанні ця програма генерує такі результати:



Можна також перевантажувати унарні оператори таки, як "++", "–", або унарні "–" і "+". При перевантаженні унарних оператора за допомогою функції-члена операторної функції жоден об'єкт не передається явно. Операція ж виконується над об'єктом, який генерує виклик цієї функції через неявно переданий покажчик this.

Розглянемо приклад:

```
// Перевантаження префіксної версії оператора "++".
class Vector
{
 double x, y;
public:
 Vector(double x = 0, double y=0)
 {
       this->x = x;
       this->y = y;
 }
 // Перевантаження префіксної версії оператора "++".
 Vector operator ++ ()
 {
       x++; // інкремент координат x, y
       y++;
       return *this;
 }

 // Перевантаження постфіксної версіяї оператора "++".
 Vector operator ++ (int notused)
 {
       Vector temp = *this; // збереження вихідного значення
       x++; // інкремент координат x, y
       y++;
       return temp; // повернення початкового значення
```

```
 }
 void Show()
 {
       cout << "x=" << x << ", y=" << y << endl;
 }
};

int main()
{
 Vector c(10, 20);
 // префіксна форма інкремента
 Vector a = ++c; // Об'єкт a отримує значення об'єкта c після його
 //інкрементування.
 a.Show(); // Тепер об'єкти a й c
 c.Show(); // мають однакові значення.
 // Постфіксна форма інкремента
 a = c++; // Об'єкт a отримує значення об'єкта c до його
 //інкрементування.
 a.Show(); // Тепер об'єкти a й c
 c.Show(); // мають різні значення.
}
```

Оператори інкременту і декременту мають як префіксну, так і постфіксну форми.

Параметр notused не використовується самою функцією. Він служить індикатором для компілятора, що дозволяє відрізнити префіксну форму оператора інкременту від постфіксної. (Цей параметр також використовується в якості ознаки постфіксної форми і для оператора декременту.)

Перевантаження операторів відношення і логічних операторів

Оператори відношення (наприклад, == або <) і логічні оператори (наприклад, && або | |) також можна перевантажувати, причому робити це

зовсім неважко. Як правило, перевантажена операторна функція відношення повертає об'єкт класу, для якого вона перевантажується. А перевантажений оператор відношення або логічний оператор повертає одне з двох можливих значень: true або false. Це відповідає звичайному застосуванню цих операторів і дозволяє використовувати їх в умовних виразах.

Розглянемо приклад перевантаження оператора "==" для класу Vector.

```
// Перевантаження оператора "=="
bool operator == (Vector op2)
{
if ((x == op2.x) && (y == op2.y))
return true;
else
return false;
}
```

Якщо вважати, що операторна функція operator == () вже реалізована, наступний фрагмент коду абсолютно коректний.

```
Vector a, b;
// ...
if(a == b)
 cout << "a дорівнює b";
else
 cout << "a не дорівнює b";
```

Ряд операторів перевантажуються парами. Наприклад, якщо ми перевантажуємо оператор ==, необхідно також перевантажити і оператор !=. А для перевантаження оператора < треба також перевантажити функцію для оператора >. І якщо для C++ це правило носить рекомендаційний характер, то в C# недотримання цього правила призведе до помилки компіляції.

Перевантаження оператора індексації масивів ([])

Крім традиційних операторів C++ дозволяє перевантажувати і більш «екзотичні», наприклад, оператор індексації масивів []. У C++ (з точки зору

механізму перевантаження) оператор [] вважається бінарним. Його можна перевантажувати тільки для класу і тільки з використанням функції-члена. Ось як виглядає загальний формат операторної функції-члена operator [ ] ().

тип operator [] (int індекс)

{

// ...

}

Припустимо, у нас визначено об'єкт ob, тоді вираз

`ob[3];` перетвориться в наступний виклик операторної функції operator []():

```
ob.operator [] (3);
```

Іншими словами, значення виразу, заданого в операторі індексації, передається операторної функції operator [] () в якості явно заданого аргументу. При цьому покажчик this буде вказувати на об'єкт ob, тобто об'єкт, який генерує виклик цієї функції.

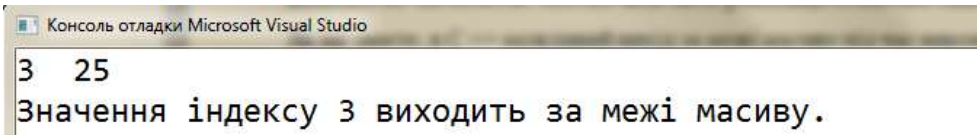
Наприклад, наступна програма оснащена засобом контролю попадання в допустимий інтервал.

```
// Приклад організації безпечного масиву.
const int SIZE = 3;
class SmartArray {
 int A[SIZE];
public:
 SmartArray() {
      for (int i = 0; i < SIZE; i++)
            A[i] = i+1;
 }
 // Контроль попадання в допустимий інтервал для класу SmartArray.
 int & operator [] (int i)
 {
      if (i <0 || i> SIZE - 1) {
```

```
            cout << "Значення індексу ";
            cout << i << " виходить за межі масиву.";
            exit(1);
        }
        return A[i];
    }
};

int main()
{
    setlocale(0, "Ukr");
    SmartArray ob;
    cout << ob[2]; // Відображається число 2.
    cout << "  ";
    ob[2] = 25; // Оператор "[]" стоїть в лівій частині.
    cout << ob[2] << endl; // Відображається число 25.
    ob[3] = 44; // Генерується помилка
    // оскільки значення 3 виходить за межі масиву.
}
```

При виконанні ця програма виводить такі результати.



Тут функція operator [] () повертає значення і-го елемента масиву A. Таким чином, вираз ob[2] повертає число 3, яке відображається інструкцією cout. Ініціалізація масиву a за допомогою конструктора виконується лише в ілюстративних цілях.

Одна з переваг перевантаження оператора "[]" полягає в тому, що з його допомогою ми можемо забезпечити засіб реалізації безпечної індексації масивів. Як ви знаєте, в C ++ можливий вихід за межі масиву під час виконання програми без відповідного повідомлення (тобто без генерування повідомлення

про динамічну помилку). Але якщо створити клас, який містить масив, і дозволити доступ до цього масиву тільки через перевантажений оператор індексації "[]", то можна перехопити індекс, значення якого вийшло за дозволені межі.

```
ob [3] = 44;
```

операторною функцією operator [] () перехоплюється помилка порушення межи масиву, після чого програма тут же завершується, щоб не допустити будь-яких потенційно можливих руйнувань.

При створенні власних класів завжди має сенс експериментувати з перевантаженням операторів. Механізм перевантаження операторів можна використовувати для додавання нових типів даних в середовище програмування. Це один з найбільш потужних засобів C ++.