**Лекція 18. Перевантаження функцій та операторів. Способи перевантаження та їх особливості**.

***Способи перевантаження операторів в С++***

Перевантаження операторів/операцій найбільш підходить для математичних класів, наприклад, де обробляються комплексні числа. При цьому для такого класу буде потрібно перевантажувати всі арифметичні операції, щоб забезпечити узгодженість оброблення в реальному житті. Мета перевантаження операцій забезпечити такі ж самі короткі вирази для користувацьких типів, як С++ забезпечує своїм набором для вбудованих типів. Але Дейтел в [4] надає таку поради щодо хорошого стилю програмування (8.3, 8.4):

* Перевантажуйте операції, щоб вони виконували над об’єктами класу ту ж саму функцію або близькі до неї функції, що і операції, виконувані над об’єктами вбудованих типів.
* Перед написанням програми на С++ з перевантаженими операціями зверніться до керівництва по вашому компілятору С++, щоб зрозуміти різноманітні специфічні обмеження та вимоги щодо окремих операцій.

Там же визначаються такі типові помилки програмування (8.1- 8.5):

* Спроба перевантажити операцію, заборонену для перевантаження.
* Спроба створювати нові операції.
* Спроба змінити роботу операції об’єктами вбудованого типу.

Наприклад, не можна перевизначити операцію складання двох цілих типів:

int operator +(int i, int j);

* Припущення, що перевантаження операції (такої як +) автоматично перевантажує, пов’язані з нею операції (такі як +=). Операції можна перевантажувати тільки явно, неявного перевантаження не існує.
* Для забезпечення узгодженості зв’язаних операцій використовуйте одні з них для реалізації інших (тобто використовуйте перевантажену операцію + для реалізації перевантаженої операції +=).

Також в [4] надаються такі зауваження по техніці програмування (8.1-8.2):

* Перевантаження операцій сприяє розширюваності С++, є однією з привабливих властивостей мови.
* Хоча б один аргумент функції-операції повинен бути об’єктом класу або посиланням на об’єкт класу. Це убезпечує програміста від зміни роботи операції з об’єктом вбудованого типу.

Функція-операція повинна визначатися або як функція-член класу, або як зовнішня функція, але дружня класу. Приклади.

Функція- член класу

**class String  
{  
  ...  
public:  
  String operator + (const String &);  
  ...  
};**

Дружня функція

**class String  
{  
  ...  
public:  
  friend String operator +(String &, String &);  
  ...  
};**

***Перевантаження унарної операції.***

Декларація функції-члену класу для перевантаження унарної операції:

**<тип\_результату> operator@ ();**

При цьому операнд операції передається неявно у вигляді вказівника **this**, тобто обов’язково має тип классу. Для реалізації операції виду **@a** відбувається виклик операторної функції: **a.operator@ ()**

Декларація дружньої функції для перевантаження унарної операції:

**friend <тип\_результату> operator@**

**(<тип\_операнду>);**

В цьому випадку для реалізації операції виду **@a** відбувається виклик операторної функції:  **operator@ (a)**

Якщо унарна операція перевантажується як функція-член, то вона не повинна мати аргументів, оскільки в цьому випадку їй передається неявний аргумент-покажчик **this** на поточний об’єкт.

Якщо унарна операція перевантажується дружньою функцією, то вона повинна мати один аргумент – об’єкт, для якого вона виконується.

Таким чином, для будь-якої унарной операції @ aa@ або @aa може інтерпретуватися або як aa.operator@(), або як operator @(aa). Якщо визначені обидві, то і aa@, і @aa є помилками.

Приклади.

Функція- член класу

**class А  
{  
  ...  
public:  
  A operator !();  
  ...  
};**

Дружня функція

**class A  
{  
  ...  
public:  
  friend A operator !(A);  
  ...  
};**

***Перевантаження бінарної операції:***

Декларація функції-члену класу для перевантаження бінарної операції:

**<тип\_результату> operator@(< тип\_операнду\_2>);**

При цьому перший операнд операції передається неявно у вигляді вказівника **this**, тобто обов’язково має тип классу. Таким чином, для реалізації операції виду **a@b**, де **a** та **b** – екземпляри відповідного класу, відбувається виклик операторної функції: **a.operator@ (b)**

Декларація дружньої функції для перевантаження бінарної операції:

**friend <тип\_результату> operator@**

**(<тип\_операнду\_1>, < тип\_операнду\_2>);**

При цьому для реалізації операції виду **a@b** відбувається виклик операторної функції: **operator@ (a,b)**

Якщо бінарна операція перевантажується з використанням метода класу, то в якості свого першого аргументу вона отримує неявно передану змінну класу ([покажчик **this**](https://prog-cpp.ru/cpp-this/) на об’єкт), а в якості другого — аргумент зі списку параметрів. Тобто, фактично бінарна операція, перевантажена методом класу, має один аргумент (правий операнд), а лівий передається неявно через покажчик this. Наприклад:

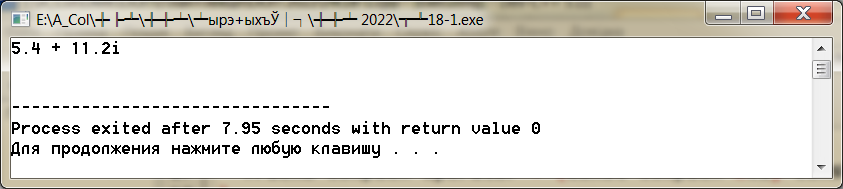
**class complex  
{  
  double real;  
  double imag;  
public:  
  complex operator +(const complex &);  
  ...  
};  
complex complex :: operator +(complex &c) {  
  complex temp;  
  temp.real = this->real + c.real;  
  temp.imag = this->imag + c.imag;  
  return(temp);  
}**

Якщо бінарна операція перевантажується дружньою функцією, то в списку параметрів вона повинна мати обидва аргументи, наприклад:

**#include <iostream>  
using namespace std;**

**#include <Windows.h>  
class complex  
{  
  double real;  
  double imag;  
public:  
  complex(double r = 0, double i = 0)  
  {  
    real = r; imag = i;  
  }  
  void out(void)  
  {  
    cout << real << " + " << imag << "i" << endl;  
  }  
  friend complex operator + (const complex &c1,**

**const complex &c2);  
};  
complex operator + (const complex &c1, const complex &c2)  
{  
  complex temp;  
  temp.real = c1.real + c2.real;  
  temp.imag = c1.imag + c2.imag;  
  return(temp);  
}  
int main()   
{ system("color F0");  
  complex a(3.1, 4.5), b(2.3, 6.7); // ініціалізація  
  complex c;  
  c = a + b;  
  c.out();  
  cin.get();  
  return 0;  
}**

****

Для кожної комбінації типів операндів в операції, що перевантажується/ перевизначається необхідно ввести окрему функцію, оскільки компілятор не може виконувати перестановку операндів місцями, навіть якщо базова операція допускає це. Наприклад, якщо необхідна операція складання комплексного та дійсного чисел:

**complex a, c, d;  
double b;  
c = a + b;  
d = b + a;**

Необхідно перевизначити операцію складання двічі:

**friend complex operator + (complex, double);  
friend complex operator + (double, complex);**

**Перевантаження операцій індексування та виклику функції**

Перевизначення операції **()** дозволяє використовувати синтаксис виклику функції стосовно об’єкта класу (ім’я об’єкта с круглими дужками). Кількість операндів в дужках може бути будь-якою.

Перевизначення операції **[]**дозволяє використовувати синтаксис доступу до елементів масиву (ім’я об’єкт с квадратними дужками).

Наданий нижче приклад ілюструє використання перевантаження операцій індексування та виклику функції.

**//-------------------------------------------------------**

**#include <iostream>**

**#include <string>**

**#include <Windows.h>**

**using namespace std;**

**//-------------------------------------------------------**

**class String // Рядок змінної довжини**

**{**

**char \*str; // Динамичний масив символів**

**int size; // Довжина рядка**

**public:**

**String & operator()(int, int); // Операція виділення підрядка**

**char operator[](int); // Операція виділення символу**

**void print() { if (str) cout << "Str = " << str << endl; }**

**friend int Find(String &,char,int); //**

**String(char\* s = (char\*)"") //**

**{**

**size = strlen(s);**

**str = new char[size + 1];**

**strcpy(str, s);**

**}**

**String(String& r) //**

**{**

**str = new char[r.size];**

**strcpy(str, r.str);**

**size = r.size;**

**}**

**};**

**//------ Операція виділення підрядка -------------------**

**String& String::operator()(int n1, int n2)**

**{ /\* n1 - ?,**

**n2 - ? \*/**

**size = n2 - n1 + 1;**

**char \*tmp = new char[size + 1];**

**for (int i = 0; i < size; i++)**

**tmp[i] = str[n1 + i];**

**tmp[size] = '\0'; //**

**size++;**

**delete[] str;**

**str = new char[size];**

**strcpy(str, tmp); //**

**delete[] tmp;**

**return (\*this);**

**}**

**//------ Операція виділення символу -------------------**

**char String::operator[](int index)**

**{**

**return (str[index]);**

**}**

**int Find(String & MyStr,char Symb,int Pos)**

**/\* MyStr - ?,**

**Symb - ?,**

**Pos - ?**

**\*/**

**{ for (int i = Pos; i < MyStr.size; i++)**

**if (MyStr.str[i]==Symb) return i; //**

**return 0;**

**};**

**int main()**

**{ system("color F0");**

**int SymbolPos; // Для символу розділювача #**

**int BegS = 0; //**

**String s1((char\*)"ab#def#ghi");**

**SymbolPos = Find(s1,'#',1); // Знайти перше входження #**

**s1.print();**

**cout << "SymbolPos = " << SymbolPos << endl;**

**String s2 = s1(BegS, SymbolPos); //**

**s2.print();**

**s1.print();**

**char ch = s2[1];**

**cout << "ch = " << ch << endl;**

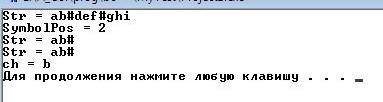
**system("pause");**

**return 0;**

**}**

**//-------------------------------------------------------**

Результат виконання

****

**Перевантаження операції присвоювання**

Будь-який конструктор викликається явно або неявно у випадку, коли необхідно створити новий об’єкт будь-якого класу.

В попередньому прикладі при створенні нового об’єкту та його ініціалізації, ми використовували конструктор копіювання. До ініціалізації існував тільки один об’єкт. Другий був створений та ініціалізований в результаті роботи конструктора копіювання. Але, якщо б існувало декілька об’єктів одного типу та була б потреба в присвоюванні значень одного об’єкта елементам іншого, то жоден з конструкторів не викликається, оскільки об’єкт вже створений. При виконанні операції присвоювання по замовчуванню копіювання значень виконується «поверхнево», але таке копіювання не завжди допустимо. Наприклад, недопустимо копіювання масивів або покажчиків.

**Особливості використання операції присвоєння.**

1. За умовчанням операція **=** для двох екземплярів класу здійснює поелементне копіювання даних-членів цих екземплярів. Якщо це саме те, що потрібно для вашого класу, немає необхідності у перевантаженні операції **=** .

2. Якщо членом класу є вказівник, то за умовчанням відбуватиметься копіювання відповідних вказівників, а не об'єктів, на які вони посилаються (чого, скоріше за все, ви очікуєте при присвоєнні). В такому разі необхідно коректно перевантажити операцію **=** .

3. Використання параметром такої операторної функції посилання на об'єкт позбавить від створення та знищення у стеку його копії і зекономить ресурси.

Якщо необхідно виконати присвоювання, але поведінка операції присвоювання по замовчуванню не влаштовує, то операція присвоювання може бути перевантажена.

Попередній приклад реалізації класу String можна доповнити перевантаженим оператором присвоювання:

**class String     // Рядок змінної довжини  
{  
  ...  
  String& operator=(const String& s)  
  {  
    if (size != s.size) {  
      size = s.size;  
      delete[] str;  
      str = new char[size+1];  
    }  
    strcpy\_s(str, size+1, s.str);  
    return \*this;  
  }  
};  
...  
int main()  
{…**

**String s1((char\*)"ab#def#ghi");  
  s1.print();  
  String s3;  
  s3 = s1;  
  s3.print();  
…**

**system("pause");**

**return 0;**

**}**

**Перевантаження операцій інкременту та декременту**

Існує нюанс визначення унарних операцій. Як відомо, і інкремент, і декремент можуть префіксними (наприклад, ++x, --y) або постфіксними (наприклад, x++, y--). Оскільки оператори інкремента та декремента є унарними і змінюють свої операнди, то перевантаження потрібно робити через методи класу. Мова С++ надає можливість реалізувати префіксні та постфіксні операції по-різному. Для цього в операторній функції, що реалізує постфіксні операції, використовується фіктивний параметр. Його наявність дозволяє перевантажити відповідним чином операції:

**// префіксний інкремент:**

**<тип\_результату> operator++ ();**

**// постфіксний інкремент:**

**<тип\_результату> operator++ (int unused);**

Або, якщо операції визначаються з допомогою дружніх функцій:

**// префіксний інкремент:**

**friend <тип\_результату> operator++ (<тип\_операнду>);**

**// постфіксний інкремент:**

**friend <тип\_результату> operator++**

**(<тип\_операнду>, int unused);**

Абсолютно аналогічні правила діють і для операції декременту.

Приклад префіксного інкремента і декремента.

**#include <iostream>**

**#include <Windows.h>**

**using namespace std;**

**class Number**

**{**

**private:**

**int m\_number;**

**public:**

**Number(int number=0)**

**: m\_number(number) { }**

**Number& operator++();**

**Number& operator--();**

**friend std::ostream& operator<< (std::ostream &out, const Number &n);**

**};**

**Number& Number::operator++()**

**{ // якщо значення змінної m\_number є 8, то встановлюємо 0**

**if (m\_number == 8)**

**m\_number = 0;**

**// Інакше просто збільшуємо m\_number на одиницю**

**else**

**++m\_number;**

**return \*this;**

**}**

**Number& Number::operator--()**

**{ // якщо значення змінної m\_number є 0, то присвюємо m\_number значенння 8**

**if (m\_number == 0)**

**m\_number = 8;**

**// Інакше просто зменшуємо m\_number на одиницу**

**else**

**--m\_number;**

**return \*this;**

**}**

**std::ostream& operator<< (std::ostream &out, const Number &n)**

**{ out << n.m\_number;**

**return out;**

**}**

**int main()**

**{ system("color F0");**

**Number number(7);**

**std::cout << number<< endl;**

**std::cout << ++number<< endl;**

**std::cout << ++number<< endl;**

**std::cout << --number<< endl;**

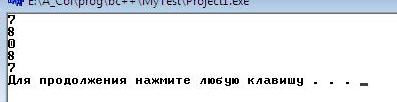
**std::cout << --number<< endl;**

**system("pause");**

**return 0;**

**}**

Результат виконання:



Зверніть увагу, що повертаємо прихований покажчик **\*this** в функціях перевантаження операторів (тобто поточний об’єкт класу Number). Таким чином, можна зв’язати виконання декількох операторів в один «ланцюжок».

Приклад постфіксного інкремента і декремента.

C++ використовує «**фіктивну змінну**» (або «**фіктивний параметр**») для операторів версії постфікс. цей фіктивний цілочисельний параметр використовується тільки з однією метою: розрізняти версію постфікс операторів інкремента/декремента від версії префікс. Виконаємо перевантаження **операторів інкремента/декремента версії префікс та постфікс в одному класі**:

**#include <iostream>**

**#include <Windows.h>**

**using namespace std;**

**class Number**

**{**

**private:**

**int m\_number;**

**public:**

**Number(int number=0)**

**: m\_number(number) { }**

**Number& operator++(); // версія префікс**

**Number& operator--(); // версія префікс**

**Number operator++(int); // версія постфікс**

**Number operator--(int); // версія постфікс**

**friend std::ostream& operator<< (std::ostream &out, const Number &n);**

**};**

**Number& Number::operator++()**

**{**

**// кщо значення змінної m\_number є 8, то встановлюємо 0**

**if (m\_number == 8)**

**m\_number = 0;**

**// Інакше просто збільшуємо m\_number на одиницю**

**else**

**++m\_number;**

**return \*this;**

**}**

**Number& Number::operator--()**

**{ // якщо значення змінної m\_number є 0, то присвюємо m\_number значенння 8**

**if (m\_number == 0)**

**m\_number = 8;**

**// Інакше просто зменшуємо m\_number на одиницу**

**else**

**--m\_number;**

**return \*this;**

**}**

**Number Number::operator++(int)**

**{ // Створюємо тимчасовий об"єкт класу Number з поточним значенням змінної m\_number**

**Number temp(m\_number);**

**// Використовуємо оператор інкремента версії префікс для реалізації перевантаження оператора инкремента версії постфікс**

**++(\*this); // реалізація перевантаження**

**// Повертаємо тимчасовий об"єкт**

**return temp;**

**}**

**Number Number::operator--(int)**

**{**

**// Створюємо тимчасовий об"єкт класу Number з поточним значенням змінної m\_number**

**Number temp(m\_number);**

**// Використовуємо оператор декремента версії префікс для реализації перевантаження оператора декремента версии постфикс**

**--(\*this); // реалізація перевантаження**

**// Повертаємо тимчасовий об’єкт**

**return temp;**

**}**

**std::ostream& operator<< (std::ostream &out, const Number &d)**

**{**

**out << d.m\_number;**

**return out;**

**}**

**int main()**

**{ system("color F0");**

**Number number(6);**

**std::cout << number<< endl;**

**std::cout << ++number<< endl; // викликається Number::operator++();**

**std::cout << number++<< endl; // викликається Number::operator++(int);**

**std::cout << number<< endl;**

**std::cout << --number<< endl; // викликається Number::operator--();**

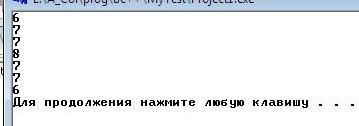
**std::cout << number--<< endl; // викликається Number::operator--(int);**

**std::cout << number<< endl;**

**system("pause");**

**return 0; }**

Результат виконання:



Зауваження до програми.

1. Версія постфікс відділена від версії префікс використанням цілочисельного фіктивного параметра в версії постфікс.
2. Оскільки фіктивний параметр не використовується в реалізації самого перевантаження, то йому навіть не надається ім’я. Таким чином, компілятор буде розглядати цю змінну як заглушку (заповнювач місця) і не буде попереджувати про створення змінної, яку не використовують.
3. Оператори версій префікс та постфікс виконують однакове завдання: обидва збільшують/ зменшують значення змінної об’єкту. Різниця між ними тільки в значенні, яке вони повертають. Оператори версії префікс повертають об’єкт після того як він був збільшений/ зменшений. В версії постфікс об’єкт повертають до того, як він був збільшений/ зменшений. Для цього використовується рішення – створити тимчасовий об’єкт з поточним значенням змінної-члена, значення якого повертають, а саму змінну-член збільшують/ зменшують. Зверніть увагу, що повернення значення за посиланням неможливо, оскільки ми ліквідуємо тимчасовий локальний об’єкт після завершення виконання тіла функції. Окрім того, це означає, оператори версії постфікс зазвичай менш ефективні, ніж оператори версії префікс, із-за додаткових витрат ресурсів на створення тимчасового об’єкта та виконання повернення по значенню замість повернення по посиланню. В прикладі реалізовано перевантаження операторів версії постфікс через вже перевантажені оператори версії префікс. Таким чином, скорочено дублювання коду та спрощено внесення змін в клас в майбутньому (тобто спрощено підтримку коду).

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Контрольні запитання для самоперевірки*.

## Що таке перевантаження операцій?

## Які операції можуть бути перевантажені?

## Які операції не можуть бути перевантажені?

## Для чого використовується ключове слово operator?

## [Що таке **унарні** та **бінарні оператори**?](https://www.bestprog.net/uk/2019/02/11/operator-overloading-in-c-operator-function-keyword-operator-overload-of-basic-arithmetic-operators-ua/#q01)

## [Які **обмеження** накладаються на перевантажені оператори?](https://www.bestprog.net/uk/2019/02/11/operator-overloading-in-c-operator-function-keyword-operator-overload-of-basic-arithmetic-operators-ua/#q08)

1. Чи можна реалізувати дві і більше операторних функцій в класі, які перевантажують однаковий оператор і отримують різні (відмінні між собою) параметри?

*Контрольні запитання для надання письмових відповідей (2 бали)*.

1. Розгляньте приклад, який ілюструє використання перевантаження операцій індексування та виклику функції. Заповніть відмічені червоним коментарі **//** та **/\* \*/*.***
2. Чи Ви можете використати цей приклад для розбору виведеного в файл рядка в Вашому застосунку з класами Person та Student для формування екземплярів класу? Якщо "так", то чи всі функції Вам будуть потрібні, якщо не потрібні, то які?

Результати надсилати на електронну адресу викладача

[**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді текстового файлу та cpp-файлу з іменем у форматі **ООП<Номер групи>-Л№18-<Прізвище англійською> та окремо надати скріншот виконання**.

**Строк відсилки ЛР ІПЗ-31 - 28.03.2024**

**ЛР ІПЗ-32 - 29.03.2024**

*Література*

1. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL:  <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>
5. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997.  URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>.