ПЕРЕВИЗНАЧЕННЯ ОПЕРАТОРІВ

## Операторні функції

Як відомо, кожен вбудований тип даних зв'язаний з визначеним набором операцій, які можна застосовувати до даних цього типу. Ці операції позначаються відповідними символами, зміст яких відомий компілятору. Програміст ніяким чином не може вплинути на виконання цих операцій.

 Коли визначається клас, то по суті визначається новий тип даних. А якщо так, то природним є бажання зробити "класові" типи максимально наближеними за можливостями до стандартних типів. Зокрема, перевизначати (перевантажувати) стандартні операції для об'єктів новостворених класів.

*Перевизначення операції* означає встановлення правила її виконання для конкретного класу. Наприклад, клас, який визначає зв'язний список, може використовувати операцію додавання "+" для внесення об'єкта до списку; клас, який реалізує стек, - для запису об'єкта в стек; у іншому класі аналогічна операція може слугувати для іншої мети.

При перевизначенні операції її вихідне значення не втрачається. Просто вводиться операція зі схожим змістом для об'єктів нового класу.

Перевантаження операцій дозволяє перетворити складний і малозрозумілий лістінг в інтуїтивно зрозумілий. Наприклад, можна реалізувати дії з екземплярами класу як звертання до певних функцій класу:

d3.addobjects(d1,d2); або d3=d1.addobjects(d2).

Проте, значно зручніше оперувати більш читабельними виразами на кшталт:

d3=d1+d2;

Перевантаженими можуть бути не всі операції С++. Оператори, дозволені до перевантаження:

+ - \* / % ^ & | ~ ! = < > += -= \*= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- ->\* , -> [] () new new[] delete delete[]

Деякі оператори заборонені до перевантаження, оскільки зміна їх змісту зруйнувала б логіку програми. До таких операторів належать: ″::" (оператор дозволу області видимості), ″.″ (оператор доступу до члена класу), ″?:″ (тернарний оператор), ″.\*″ (доступ до розіменованого вказівника-члена класу), sizeof. Крім того, не рекомендується перевантажувати логічні оператори && і ||, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів (якщо на деякому етапі значення усього виразу стає визначеним, подальші обчислення припиняються).

Перевизначення операторів здійснюється за допомогою так званих *операторних функцій* - функцій із спеціальним ім'ям operator@, де @ - символ операції, що перевизначається. Загальний формат операторної функції є таким:

тип ім'я\_класу::operator@(параметри)

{ тіло

}

Тіло операторної функції власне і визначає нову дію відповідного оператора. Елемент тип вказує на тип значення, що повертається внаслідок виконання даної операції. І хоча він може бути будь-яким, частіше за все тип значення збігається з іменем класу, для якого перевизначається функція operator(). Такий зв'язок полегшує використання перевизначеного оператора у складних арифметичних і логічних виразах.

Механізм перевизначення операторів тісно пов'язаний з механізмом перевизначення функцій. Компілятор може розрізняти перевантажувані функції за типом даних та кількістю аргументів (сигнатурою). Перевантажувані операції компілятор розрізняє за типом даних їх операндів. Якщо операнд має стандартний базисний тип, такий як, наприклад, int, то компілятор буде використовувати вбудовану функцію для обробки змінної типу int. Але, якщо операнд є об’єктом класу, то компілятор буде використовувати написану програмістом функцію operator().

Для багатьох операцій існують свої особливості при перевантаженні. Так,

* Операція «=» може бути використана будь-яким класом без явного перевантаження. За замовчуванням операція присвоювання зводиться до побітового копіювання атрибутів класу. Таке копіювання небезпечне для класів з атрибутами, які вказують на динамічно виділені області пам'яті, тому для таких класів варто явно перевантажувати цю операцію.
* Операція адресації «&» також може бути використана з об'єктами будь-яких класів без перевантаження, вона просто повертає адресу об'єкта в пам'яті.

Інші обмеження, що супроводжують застосування перевантажених операторів:

1. Перевантажені функції не можуть змінювати пріоритет і асоціативність операторів (порядок виконання операцій одного пріоритета).

2. Кількість операндів операції фіксована.

3. Значення операндів не можна задавати за замовчуванням.

Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями. Операторні функції-члени і не члени класу відрізняються між собою механізмом перевизначення

## Перевизначення операторів з використанням методів класу

Як відомо, оператори можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний — два. До унарних операторів, що можуть перевантажуватися, належать такі операції, як "+", "-", "++", "--", "&", "~" і "!"; до бінарних - всі інші оператори, перераховані вище.

### **Перевизначення унарних операторів**

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: такі функції не мають явних параметрів. Операція здійснюється над об'єктом, який передається операторній функції неявно за допомогою вказівника this.

Операторна функція може повертати об'єкт класу (точніше, розіменований вказівник this) чи посилання на об'єкт, а може і нічого не повертати. Вибираючи тип значення, що повертається, необхідно керуватися здоровим глуздом. Якщо унарний оператор повинен використовуватися усередині виразів, то операторна функція повинна повертати об'єкт чи посилання; якщо ж оператор використовується ізольовано, функція може нічого не повертати.

Варто мати на увазі, що хоча зміст оператора перевантажувати можна, його природу змінювати заборонено. Це означає, що унарні оператори не можна перевантажувати як бінарні і навпаки.

Перевизначення унарних операцій “+” і “-”. Розглянемо приклад перевантаження унарного “-” (операція унарного “+” є “порожньою” і включається в мову програмування тільки для симетрії).

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& z) { Re = z.Re; Im = z.Im; }

~TComplex() { }

void print();

TComplex operator-() { Re = -Re; Im = -Im; return \*this;}

};

int main()

{ TComplex z(1,1), u(0,0);

z.print();

u=-z;

u.print();

getch();

}

void TComplex::print()

{ printf("%lf + i\*%lf\n", Re, Im);

}

Рядок програми

u = - z;

можна переписати в еквівалентному вигляді

u = z.operator-();

Цей рядок демонструє, що виклик операторної функції operator-() здійснюється об'єктом z. Однак лістінг програми у цьому випадку стає менш зрозумілим.

Перевизначення унарних операцій інкремента "++" та декремента "--". Ще одним засобом підвищення наочності програми є перевизначення унарних операцій інкремента "++" та декремента "--". Як відомо, дані оператори можуть мати дві форми: префіксну (++ObjC) та постфіксну (ObjC++).

Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій — вводиться фіктивний цілочисельний параметр. Наприклад, для класу TComplex прототипи відповідних операторних функцій можуть виглядати в такий спосіб:

TComplex operator++(); // префіксна форма

TComplex operator++(int notused); // постфіксна форма

TComplex operator--(); // префіксна форма

TComplex operator--(int notused); // постфіксна форма

Тут фіктивний цілочисельний параметр notused не грає ролі аргументу і не означає цілого числа. Це просто сигнал для компілятора, щоб використовувалася постфіксна версія операції.

Наприклад,

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& z) { Re = z.Re; Im = z.Im; }

~TComplex() { }

void print();

TComplex operator++();

TComplex operator++(int notused);

TComplex operator--();

TComplex operator--(int notused);

};

int main()

{ TComplex z(1,1);

++z;

z.print();

z++;

z.print();

--z;

z.print();

z--;

z.print();

getch();

}

void TComplex::print()

{ printf("% lf + i\*%lf\n", Re, Im);

}

TComplex TComplex::operator++()

{ ++Re;

++Im;

printf("Prefix++ \n");

return \*this;

}

TComplex TComplex::operator--()

{ --Re;

--Im;

printf("Prefix-- \n");

return \*this;

}

TComplex TComplex::operator++(int notused)

{ ++Re;

++Im;

printf("Postfix++ \n");

return \*this;

}

TComplex TComplex::operator--(int notused)

{ --Re;

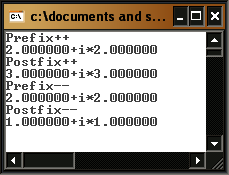
--Im;

printf("Postfix-- \n");

return \*this;

}

Внаслідок виконання ця програма відображає на екрані такі результати:



Якщо символ операції ″++″ стоїть перед операндом, викликається операторна функція operator++(), якщо після — операторна функція operator++(int notused). Змінна i відіграє роль прапорця, який повідомляє компілятору, що дана функція перевантажує постфіксну форму оператора інкремента і декремента.

Незважаючи на те, що програміст може вільно трактувати перевантажений оператор, прагнучи зберегти аналогію з його вбудованими аналогами, необхідно дотримуватися визначених правил.

Наприклад, як відомо, у С++ оператор інкрементації цілих чисел повертає посилання на неконстантний об'єкт. Ця властивість повинна зберігатися і при перевантаженні. У той же час постфіксний оператор повертає константне значення (саме тому в С++ неможливі вирази x++++), тому цілком природним є те, що перевантажений оператор має таку ж властивість. З урахуванням вищезазначеного, операторну функцію для перевизначення постфіксного оператора інкрементації має сенс представити так:

const TComplex TComplex::operator++(int notused)

{ ++Re;

++Im;

printf("Постфіксна форма ++ \n");

return \*this;

}

Крім того, потрібно мати на увазі, що традиційний постфіксний оператор інкремента спочатку набуває значення операнда, а потім його інкрементує. Отже, перш ніж інкрементувати поточне значення операнда, його потрібно зберегти у тимчасовому об’єкті, а потім повернути (не забувайте, що постфіксний оператор інкремента не повинен повертати модифіковане значення свого операнда). Наприклад,

const TComplex TComplex::operator++(int notused)

{ TComplex tmp=\*this; // збереження початкового значення об'єкта

++Re;

++Im;

printf("Постфіксна форма ++\n");

return tmp; // повернення початкового значення об'єкта

}

Тут операторна функція для реалізації постфіксної форми унарного оператора інкремента "++" зберігає початкове значення операнда у тимчасовому об’єкті tmp наступним чином:

TComplex tmp = \*this;

Збережене (у об'єкті tmp) значення операнда повертається за допомогою оператора return.

Перевизначення унарних операцій ″!″, ″&″ і ″~″. Оператори заперечення (″!″), взяття адреси (″&″) і побітового заперечення (″~″) теж допускають перевантаження. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажімо, за допомогою оператора ″!″ можна позначати операцію обернення матриці, а за допомогою символу ″~″ — її транспонування. Щоправда, застосування символу ″тільди″ (″~″) закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанини. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.

Перевантаження оператора ″->″. Оператор посилання на член об'єкта є унарним. Операторна функція, що перевантажує його, виглядає в такий спосіб:

об'єкт -> елемент

Цей запис еквівалентний наступному виразу:

об'єкт.operator->(елемент);

Функція operator- >() повинна бути нестатичним членом класу. Як параметр, вона одержує об'єкт класу чи посилання на нього, повертаючи вказівник this на об'єкт, де виконується виклик, або посилання на об'єкт будь-якого іншого класу, у якому визначений оператор ″->″. Її зручно використовувати в контейнерних класах, що містять усередині себе вказівник на інший клас. Основний зміст перевантаження оператора ″->″ полягає в додатковій функціональності, що розширює можливості звичайних вказівників.

Наприклад, у приведеній нижче програмі функція operator- >() веде підрахунок посилань на кожен об'єкт класу.

class TClass

{ int n;

int counter;

public:

TClass(int x):n(x),counter(0) { }

TClass\* operator->();

int get(void) { return n;}

int ref(void) { return counter; }

};

TClass\* TClass::operator ->()

{ counter++;

return this;

}

int main()

{ TClass a(1), b(2);

printf("n = %d \n",a->get());

printf("n = %d \n",b->get());

printf("n = %d \n",a->get());

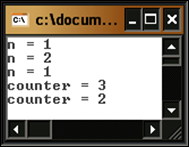
printf("counter = %d \n",a->ref());

printf("counter = %d \n",b->ref());

getch();

}

У результаті роботи програми на екран виводяться наступні рядки:



### **Перевизначення бінарних операторів**

При перевизначенні бінарної операції з використанням метода класу відповідній операторній функції безпосередньо передається тільки один аргумент, другий же - неявно передається через покажчик this. При цьому опосередковано передається об'єкт, який вказується зліва від символу операції; об'єкт, який розташовується з правого боку від символу операції, передається цій функції як аргумент.

Таким чином, бінарна операторна функція-член має тільки один параметр, що задає другий операнд. Вказівник this на перший операнд вона одержує неявно.

Розглянемо приклади перевантаження арифметичних операцій, операцій порівняння та операції присвоєння.

Перевантаження арифметичних операцій. Реалізуємо перевантаження арифметичних операції для класу комплексних чисел:

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& obj) { Re = obj.Re; Im = obj.Im; }

~TComplex() { }

void print(char \*str);

TComplex TComplex::operator+(const TComplex obj);

TComplex TComplex::operator-(const TComplex obj);

TComplex TComplex::operator\*(const TComplex obj);

TComplex TComplex::operator~();

TComplex TComplex::operator/(TComplex obj);

double TComplex::modul2(const TComplex obj);

};

int main()

{ TComplex ObjA(1,1), ObjB(2,2), ObjC(0,0), ObjD(0,0);

ObjC=ObjA+ObjB; // додавання об'єктів ObjA і ObjB

ObjC.print(“ObjC”);

ObjD=ObjC\*ObjA/ObjB;

ObjD.print(“ObjD”);

getch();

}

void TComplex::print(char \*str)

{ printf("%s=%lf + i\*%lf\n", str, Re, Im);

}

TComplex TComplex::operator+(const TComplex obj) // додавання

{ TComplex tmp (0,0);

tmp.Re = Re+obj.Re;

tmp.Im = Im+obj.Im;

return tmp;

}

TComplex TComplex::operator-(const TComplex obj) // віднімання

{ TComplex tmp(0,0);

tmp.Re = Re-obj.Re;

tmp.Im = Im-obj.Im;

return tmp;

}

TComplex TComplex::operator\*(const TComplex obj) // множення

{ TComplex tmp(0,0);

tmp.Re = Re\*obj.Re-Im\*obj.Im;

tmp.Im = Re\*obj.Im+Im\*obj.Re;

return tmp;

}

TComplex TComplex::operator~() // спряжене число

{ TComplex tmp(0,0);

tmp.Re = Re;

tmp.Im = -Im;

return tmp;

}

TComplex TComplex::operator/(TComplex obj)

{ TComplex tmp(0,0);

tmp=(\*this)\*(~obj);

tmp.Re = tmp.Re/modul2(obj);

tmp.Im = tmp.Im/modul2(obj);

return tmp;

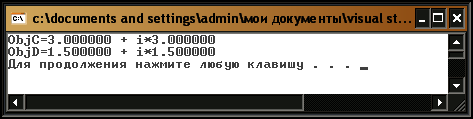
}

double TComplex::modul2(const TComplex obj)

{ return obj.Re\*obj.Re+obj.Im\*obj.Im;

}

Результат роботи програми наступний:



Як видно із прикладу, операторні функції для перевантаження арифметичних операцій мають тільки по одному параметру. Другий же опосередковано передається через покажчик this. Таким чином, у рядку

tmp.Re = Re+obj.Re;

під операндом Re мається на увазі атрибут this->Re, тобто атрибут Re зв'язується з об'єктом, який викликає дану операторну функцію.

Щоб зрозуміти механізм перевизначення операторів, розглянемо уважно наведену вище програму, починаючи з перевизначеного оператора додавання "+".

Під час обробки двох об'єктів типу TComplex оператором додавання "+" виконуються операції додавання значень відповідних комплексних чисел так, як це показано у функції operator+(). При цьому дана операторна функція не модифікує значень жодного операнда. Як результат виконання операції ця функція повертає об'єкт типу TComplex, який містить результати попарного додавання двох комплексних чисел.

У цій програмі сума комплексних чисел ObjA і ObjB присвоюється об'єкту ObjC. Виклик операторної функції operator+() здійснюється з об'єкта ObjA. При цьому бінарний оператор ObjA+ObjB еквівалентний оператору ObjA.operator+(ObjB). Але, використовуючи перезавантаження операції +, ми зробили вираз більш зрозумілим і наочним.

У виразі

ObjC=ObjA+ObjB;

важливо розуміти, до яких об’єктів відноситимуться аргументи і значення, що повертаються. Коли компілятор зустрічає цей вираз, він проглядає типи аргументів. Виявивши лише аргументи типу TComplex, він виконує операції виразу, використовуючи метод класу TComplex operator+(). При цьому об’єкт з лівої сторони операції (в нашому випадку ObjA) викликає функцію оператора. Об’єкт, що стоїть праворуч від знаку операції, передається у функцію в якості аргумента.

В функції operator+() до лівого операнда ми маємо прямий доступ, використовуючи Re та Im, оскільки це об’єкт, що викликає функцію. До правого операнда ми маємо доступ як до аргументу функції, тобто як objB.Re та obj.Im.

Оскільки сума двох об'єктів класу TComplex є об'єктом цього ж класу, не обов'язково створювати новий об'єкт і записувати в нього результат підсумовування. Можна скористатися наступною синтаксичною конструкцією:

(ObjA+ObjB).print(“ObjA+ObjB”);

У цьому випадку функція print() буде викликатися тимчасовим об'єктом, створеним операторної функцією operator+().

Цей механізм допускає вживання безглуздих виразів. Наприклад, компілятор не заперечує проти такого оператора:

(ObjA+ObjB)=ObjD;

Для того, щоб запобігти цьому, оператор ″+″ повинен повертати константний об'єкт. Крім того, бажано, щоб він не змінював другий операнд. Отже, параметр теж повинен бути константним.Отже, одержуємо наступну реалізацію:

const TComplex TComplex::operator+(const TComplex obj)

{ TComplex tmp(0,0);

tmp.Re = Re+obj.Re;

tmp.Im = Im+obj.Im;

return tmp;

}

У класі TComplex операція ділення двох комплексних чисел реалізована через обчислення спряженого числа (операція ~) і ділення дійсної і мнимої частин на квадрат модуля дільника (функція modul2()).

Для того щоб бінарну операторну функцію можна було застосовувати усередині виразів, необхідно, щоб вона повертала об'єкт свого класу. Наприклад, операторна функція operator\*() повертає об'єкт типу TComplex, що дає змогу використовувати оператор множення "\*" у таких складних виразах, як ObjC\*ObjA/ObjB. Частина цього виразу (ObjC\*ObjA) отримує результат типу TComplex, на який потім ділиться об'єкт ObjC. І якби ця частина виразу генерувала значення іншого типу (а не типу TComplex), то такий складний вираз обчислити не було б змоги.

Перевантаження операторів відношення та логічних операторів. Оператори відношення (наприклад, "==", "<", ">", "<=", ">=", "!=") і логічні оператори (наприклад, "&&" або "||") також можна перевизначати. Будь-який перевизначений оператор відношення або логічний оператор повертає одне з двох можливих значень: true або false. Це відповідає звичайному застосуванню цих операторів і дає змогу використовувати їх в умовних виразах.

Розглянемо приклад перевизначення операторної функції дорівнює "==" для класу kooClass, що визначає тривимірні координати:

class kooClass

{ intx, y, z; // тривимірні координати

public:

kooClass() { x = y = z = 0; }

kooClass(int c, int d, int f) {x = c; y = d; z = f; }

bool operator==(kooClass obj) // перевизначення операторної функції "=="

void Show(char \*s);

};

bool kooClass::operator==(kooClass obj)

{ if((x == obj.x) && (y == obj.y) && (z == obj.z)) return true;

else return false;

}

Якщо вважати, що операторна функція operator==() вже реалізована, то такий код програми є абсолютно коректним:

kooClass ObjA, ObjB;

//...

if(ObjA == ObjB) cout << "ObjA = ObjB" << endl;

else cout << "ObjA не дорівнює ObjB" << endl;

Оскільки операторна функція operator==() повертає результат типу bool, то її можна використовувати для керування розгалуженням у операторі if.

Перевантаження оператора присвоювання. У програмах, розглянутих вище, ми вільно маніпулювали об'єктами, присвоюючи їх один одному. Це відбувалося завдяки вбудованому оператору присвоювання, що виконує побітове копіювання об'єкта, який знаходиться праворуч від операції "=", в об'єкт, який знаходиться ліворуч від цієї операції.

Як і у випадку з конструктором копіювання, це — не краще рішення. Якщо об'єкт містить покажчик на деяку область пам'яті, його копія також буде посилатися на неї. Для того щоб цього не відбулося, слід перевантажити оператор присвоювання таким чином, щоб він повинен виконував глибоке копіювання, уникаючи подвійної адресації.

Щоб зрозуміти суть описаної вище проблеми, розглянемо таку (некоректну) програму:

class strClass

{ char \*s;

public:

strClass() { s = 0; }

strClass(const strClass &obj);

~strClass() { if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }

void Show(char \*c) { cout << c << s << endl; }

void Set(char \*str);

};

strClass::strClass(const strClass &obj)

{ s = new char[strlen(obj.s)+1];

strcpy(s, obj.s);

}

void strClass::Set(char \*str) // завантаження рядка

{ s = new char[strlen(str)+1];

strcpy(s, str);

}

strClass Init()

{ char str[80];

strClass obj;

cout << "Введіть рядок: "; cin >> str;

obj.Set(str);

return obj;

}

int main()

{ strClass Obj;

Obj = Init(); // цей оператор генерує помилку!!!!

Obj.Show("s= ");

getch();

}

Залежно від використовуваного компілятора, на екрані монітора буде виводитися переважно інформація про "сміття" або генеруватися помилка тривалості виконання програми. У будь-якому випадку помилки не минути. І ось чому.

У цьому коді програми конструктор копії коректно обробляє повернення об'єкта функцією Init(): коли функція повертає об'єкт, то для зберігання повернутого нею значення створюється тимчасовий об'єкт. Оскільки при створенні об'єкта-копії конструктор копії виділяє нову область пам'яті, то атрибут s початкового об'єкта і атрибут s об'єкта-копії вказуватимуть на різні області пам'яті, які, як наслідок, не стануть псувати одна одну.

Проте помилки не минути, якщо повернутий функцією об'єкт присвоюється об'єкту Obj, оскільки у процесі виконання операції присвоєння за замовчуванням створюється побітова його копія. У цьому випадку тимчасовий об'єкт, який повертається функцією Init(), копіюється в об'єкт Obj. Як наслідок, атрибут obj.s вказує на ту ж саму область пам'яті, що і атрибут s тимчасового об'єкта. Але після виконання операції присвоєння в процесі руйнування тимчасового об'єкта ця пам'ять звільняється. Отже, член obj.s після цього вказуватиме на вже звільнену пам'ять. Окрім цього, пам'ять, яка адресується членом obj.s, повинна бути звільнена і після завершення роботи програми, тобто вдруге.

Щоб запобігти цьому, необхідно перевизначити оператор присвоєння таким чином, щоб об'єкт, який розташовується зліва від оператора присвоєння, виділяв власну область пам'яті.

Реалізація цього рішення наведена у наступній програмі:

class strClass

{ char \*s;

public:

strClass();

strClass(const strClass &obj);

~strClass() { if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }

void Show(char \*c) { cout << c << s << endl; }

void Set(char \*str);

strClass operator=(const strClass &obj); // перевизначений оператор присвоєння

};

strClass::strClass()

{ s = new char ('\0');

}

strClass::strClass(const strClass &obj)

{ s = new char[strlen(obj.s)+1];

strcpy(s, obj.s);

}

void strClass::Set(char \*str) // завантаження рядка

{ s = new char[strlen(str)+1];

strcpy(s, str);

}

strClass strClass::operator=(const strClass &obj)

{ if(strlen(obj.s) > strlen(s)) // якщо пам'яті недостатньо, виділяється нова область пам'яті

{ delete[]s;

s = new char[strlen(obj.s)+1];

}

strcpy(s, obj.s);

return \*this;

}

strClass Init()

{ strClass obj; char str[80];

cout << "Введіть рядок: "; cin >> str;

obj.Set(str);

return obj;

}

int main()

{ strClass Obj;

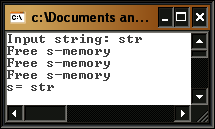
Obj = Init(); // тепер тут все гаразд!

Obj.Show("s= ");

getch();

}

Ця програма відображає такі результати:



Як бачимо, програма тепер працює коректно.

На відміну від арифметичних операторів, оператор присвоєння "=" модифікує один зі своїх аргументів. Оскільки операторна функція operator=() викликається об'єктом, який розташований зліва від символу присвоєння "=", то саме цей об'єкт і модифікується внаслідок виконання операції присвоєння. Після виконання цієї операції значення, яке повертається перевизначеним оператором, містить об'єкт, який було вказано зліва від символу присвоєння. Наприклад, щоб можна було виконувати настанови, подібні до такої (множинне присвоєння)

ObjA = ObjB = ObjC = ObjD;,

необхідно, щоб операторна функція operator=() повертала об'єкт, який адресується

покажчиком this, і щоб цей об'єкт розташовувався зліва від оператора присвоєння

"=". Це дасть змогу виконати будь-який ланцюжок присвоєнь.

Операція присвоєння – це одне з найважливіших застосувань покажчика this.

Незважаючи на зовнішню схожість перевантаженого оператора присвоювання і конструктора копіювання, між ними існує принципова різниця. При виклику конструктора копіювання створюється новий об'єкт, що ініціалізується раніше існуючим об'єктом. При присвоюванні обидва об'єкти уже існують.

Перевантаження скорочених операторів присвоювання. В арифметичних обчисленнях дуже корисні скорочені оператори присвоювання, тому має сенс спробувати перевантажити їх для знову створюваних класів.

При цьому варто враховувати обмеження, що накладаються на оператор присвоювання і арифметичні оператори. Як правило, перевантажені арифметичні оператори реалізуються за допомогою скорочених операторів присвоювання.

Наприклад,

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& obj) { Re = obj.Re; Im = obj.Im; }

~TComplex() { }

void print();

const TComplex operator+=(const TComplex& z);

const TComplex operator+(const TComplex& z);

};

int main()

{ TComplex u(1,1),v(2,2),z(3,3);

u+=v;

u.print();

z=u+v;

z.print();

getch();

}

const TComplex TComplex::operator+=(const TComplex& z)

{ Re = Re+z.Re; Im = Im+z.Im;

printf("Operator += \n");

return \*this;

}

const TComplex TComplex::operator+(const TComplex& z)

{ TComplex w=\*this;

w+=z;

printf("Operator + \n");

return w;

}

void TComplex::print()

{ printf("% lf + i\*%lf\n", Re, Im);

}

### **Перевантаження оператора послідовного обчислення “,”**

Як випливає з визначення стандартної операції послідовного обчислення, її значенням є результат обчислення останнього виразу серед перерахованих у списку. Отже, головним фактором, що впливає на результат застосування цього оператора, є порядок перерахування його операндів. Однак, коли оператор “,” перевантажується за допомогою операторної функції, операнди стають параметрами функції. Це створює проблему, тому що порядок обчислення аргументів функції стандартом мови не визначений. Таким чином, перевантаження оператора послідовного обчислення може привести до неправильної роботи програми і непередбачених ефектів.

Продемонструємо програму, що ілюструє потенційну можливість перевантажувати оператор “,”:

class TIndex

{ int i;

public:

TIndex(int init):i(init) { }

void print() { printf("Index = %d\n", i); }

TIndex operator++() { ++i; return \*this; }

TIndex operator--() { --i; return \*this; }

TIndex operator,(TIndex last);

bool operator>(TIndex rhs)

{ if ((\*this).i > rhs.i)

return true;

else return false;

}

};

TIndex TIndex::operator,(TIndex last)

{ TIndex tmp(0);

tmp.i = last.i;

return tmp;

}

int main()

{ TIndex last(0);

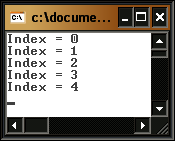
for(TIndex i(0),j(5); j>0; ++i,--j)

i.print();

getch();

}

Як і слід було очікувати, на екран виводяться такі рядки.



Лівий операнд передається операторній функції неявно — за допомогою вказівника this. Функція ігнорує його значення, повертаючи значення правого операнда

### **Перевантаження операторів new і delete**

Програміст може керувати виділенням пам'яті, перевантажуючи оператори new і delete. Перевантажена операторна функція operator new() має наступний вигляд:

void\* operator new(size\_t size);

Вона виділяє size байт пам'яті і повертає адресу виділеної пам'яті. Конструктор і деструктор об'єктів викликаються автоматично. Тип size\_t є цілочисельним.

Перевантажений оператор delete звільняє пам'ять, виділену перевантаженим оператором new.

Розглянемо програму, у якій оператор new реалізує виділення пам'яті за допомогою функції malloc(). Таке перевантаження може виявитися корисним при сполученні декількох модулів, написаних на мовах С і С++. Як відомо, одночасне використання механізмів розподілу пам'яті, передбачених у мовах С і С++, тобто використання пар new — free() чи malloc() — delete, може привести до непередбачених результатів. Перевантаження ж оператора new, зазначеним вище чином, може привести різні модулі “до загального знаменника”.

Наприклад, програма, у якій перевантажені оператори new і delete, що використовують функції malloc() і free(). У класі передбачена статична змінна — індикатор зайнятої пам'яті. Як тільки обсяг виділеної пам'яті перевищує допустимий, програма видає повідомлення на екран.

#define MAX\_SIZE 1024

class TClass

{ char array[500];

public:

TClass(){cout << "Ctor" << endl;}

~TClass(){cout << "Dtor" << endl;}

static void\* operator new(size\_t);

static void operator delete(void\*);

static long counter;

};

void\* TClass::operator new(size\_t size)

{ if( counter >= MAX\_SIZE - sizeof(TClass))

{ cout << "Пам'ять вичерпана" << endl;

return 0;

}

else

{ counter += sizeof(TClass);

cout << "New " << endl;

cout << "Зайняте: " << counter << " байт" << endl;

return malloc(size);

}

}

void TClass::operator delete(void\* p)

{ free(p);

counter-=sizeof(TClass);

cout << "Delete " << endl;

cout << "Зайнято: " << counter << " байт" << endl;

};

long TClass::counter = 0;

int main()

{ TClass\* q[3];

for (int i = 1; i<=3; i++)

{ q[i] = new TClass; }

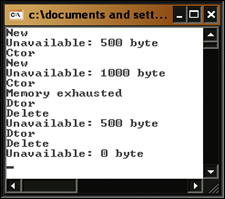
for (int i = 1; i<=3; i++)

{ delete q[i]; }

getch();

}

Ця програма веде облік зайнятої пам'яті, збільшує розмір лічильника, а потім звільняє пам'ять і зменшує лічильник на відповідну величину:



Тепер застосування операторів new і delete до об'єктів класу TClass приведе до виклику перевантажених версій, а для вбудованих типів використовуються звичайні варіанти цих операторів.

### **Перевантаження операторів new[ ] і delete[ ]**

При створенні власного механізму розподілу пам'яті для масивів, можна перевантажити оператори new[] і delete[], керуючись тими ж правилами.

Наприклад,

#include <malloc.h>

void\* operator new[](size\_t size)

{ void \*p;

cout << "New " << endl;

p = malloc(size);

return p;

}

void operator delete[](void\* p)

{ cout << "Delete\n";

free(p);

}

int main()

{ char\* q;

q = new char[5];

getch();

}

Ця програма виділяє пам'ять для масиву, що складає з 5 символів, а потім звільняє її. Як правило, перевантажені оператори new і delete генерують виняткові ситуації. Однак існують версії цих операторів, у яких генерація виняткової ситуації скасована.

### **Перевантаження оператора індексації елементів масиву "[ ]"**

Операція індексації елементів масиву "[]", яка зазвичай, використовується для доступу до елементів масиву, теж може бути перевантажена. Це корисно в тому випадку, коли ми хочемо змінити спосіб роботи С++ з масивами. Наприклад, нам може знадобитися «безпечний масив», в якому закладена автоматична перевірка використовуваного для доступу до масиву індексу елемента. Тоді при виконнні операції індексації можна перевіряти, чи не вийшли ми за межі масиву.

У C++ (з погляду механізму перевизначення) оператор доступу до члена масиву "[]" вважається бінарним. Його можна перевизначати тільки для класу і тільки з використанням методу класу.

Оператор індексації "[]" перевантажується за допомогою наступної операторної функції operator[](), яка повинна бути нестатичним членом класу:

тип [&] ім'я\_класу::operator[](int індекс)

{ //...

}

Формально параметр *індекс* необов'язково повинен мати тип int, але операторна функція operator[](), зазвичай, використовується для забезпечення індексації елементів масивів, тому в загальному випадку як аргумент цієї функції передається цілочисельне значення.

Елементи масиву можуть стояти як у лівій, так і в правій частині оператора присвоювання. Отже, функція operator[]() повинна повертати посилання або покажчик.

Операторна функція operator[]() повинна бути нестатичним методом класу.

Як зазначалося вище, функція operator[]() є бінарною. Її перший параметр явно задає індекс елемента, а другий параметр, що представляє собою вказівник this на об'єкт, де виконується виклик, передається неявно.

Нехай, наприклад, створено об'єкт ObjA. Тоді вираз ObjA[3] перетвориться в такий виклик операторної функції operator[]():

ObjA.operator[](3);

Значення виразу, що задається в операторі "[]", передається операторній функції operator[]() як безпосередньо заданий аргумент. При цьому покажчик this вказуватиме на об'єкт ObjA, тобто об'єкт, який здійснює виклик цієї функції.

У наведеній нижче програмі представлений механізм перевизначення оператора індексації елементів масиву "[]". В класі aClass оголошується масив для зберігання трьох int-значень. Його конструктор ініціалізує кожен член цього масиву. Перевизначена операторна функція operator[]() повертає значення елемента, що задається його параметром.

const int size = 3;

class aClass

{ int aMas[size];

public**:**

aClass() { for(int i=0; i<size; i++) aMas[i] = i\*i; }

int operator[](int i) {return aMas[i]; }

};

int main()

{ aClass ObjA;

cout << "aMas[2]= " << ObjA[2] << endl; // відображає число 4

cout << "Masiv<A>:" << endl;

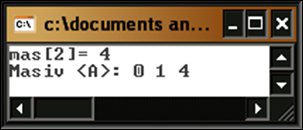
for(int i=0; i<3; i++)

cout << " " << ObjA[i] << endl;

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:



Ініціалізація масиву aMas за допомогою конструктора у цій програмі здійснюється виключно з ілюстративною метою. Функція operator[]() спочатку повертає значення 3-го елемента масиву aMas, а вираз ObjA[2] - число 4, яке відображається в потоці cout. Потім у циклі виводяться усі елементи масиву.

Можна розробити операторну функцію operator[]() так, щоб оператор індексації елементів масиву "[]" можна було використовувати як зліва, так і праворуч від оператора присвоєння. Для цього достатньо вказати, що значення, яке повертається операторною функцією operator[](), є посиланням. Цю можливість продемонстровано у наведеному нижче коді програми:

const int size = 3;

class aClass

{ int aMas[size];

public**:**

aClass() { for(int i=0; i<size; i++) aMas[i] = i\*i; }

int &operator[](int i) {return aMas[i]; }

};

int main()

{ aClass ObjA;

cout << " Masiv<A>:" << endl;

for(int i=0; i<3; i++)

cout << "aMas[" << i << "]= " << ObjA[i] << endl;

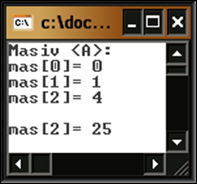
ObjA[2] = 25; // оператор "[]" знаходиться зліва від оператора "="

cout << endl << "aMas[2]= " << ObjA[2]; // тепер відображається число 25

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:



Оскільки операторна функція operator[]() тепер повертає посилання на елемент масиву, що індексується параметром i, то оператор індексації елементів масиву "[]" можна використовувати зліва від оператора присвоєння, що дає змогу модифікувати будь-який елемент масиву.

Одна з наявних переваг перевизначення оператора індексації елементів масиву "[]" полягає у тому, що за допомогою нього можна забезпечити реалізацію безпечної індексації елементів масиву. Як зазначалося вище, у C++ можливий вихід за межі масиву у процесі виконання програми без генерування відповідного повідомлення. Але, якщо створити клас, який містить масив, і надати доступ до цього масиву тільки через перевизначений оператор індексації елементів масиву "[]", то в процесі виконання програми можливе перехоплення індексу, значення якого вийшло за дозволені межі. Наприклад, наведений нижче код програми містить засоби контролю потрапляння індексу масиву в допустимий інтервал його перебування:

const int size = 3;

class aClass

{ int aMas[size];

public**:**

aClass() { for(int i=0; i<size; i++) aMas[i] = i\*i; }

int &operator[](int i);

};

int &aClass::operator[](int i) // контроль потрапляння індексу масиву в допустимий інтервал

{ if ( i<0 || i > size-1)

{ cout <<endl<<"Значення індексу "<< i <<" виходить за межі допустимого інтервалу"<<endl;

getch(); exit(1);

}

return aMas[i];

}

int main()

{ aClass ObjA;

cout << "Значення елементів масиву <A>:" << endl;

for(int i=0; i<3; i++)

cout << "aMas[" << i << "]= " << ObjA[i] << endl;

ObjA[2] = 25; // оператор "[]" знаходиться в лівій частині

cout << endl << "aMas[2]= " << ObjA[2]; // відображається число 25

ObjA[3] = 44; // помилка, оскільки значення індексу виходить за межі допустимого інтервалу

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Значення елементів масиву <A>:

a[0]= 0

a[1]= 1

a[2]= 4

a[2]= 25

Значення індексу 3 виходить за межі масиву.

У процесі виконання оператора

ObjA[3] = 44;

операторною функцією operator[]() перехоплюється помилка порушення меж допустимого інтервалу перебування індексу масиву, після чого програма відразу завершується, щоб не допустити потім ніяких потенційно можливих руйнувань.

### **Перевантаження оператора виклику функцій "()"**

Можливо, найбільш цікавим оператором, який можна перевизначати, є оператор виклику функції "()". Під час його перевизначення створюється не новий спосіб виклику функцій, а операторна функція, якій можна передати довільну кількість параметрів. Також ця функція може повертати значення будь-яких типів.

Об'єкти, що містять операторну функцію operator()(), називаються функціями-об'єктами або *функторами*. Такі об'єкти бувають корисними при виконанні операцій, зв'язаних з декількома індексами.

Операторна функція operator()() повинна бути нестатичним методом класу.

Припустимо, що певний клас містить наведене нижче оголошення перевизначеної операторної функції:

int operator()(float f, char \*p);

Тоді, якщо у програмі створюється об'єкт obj цього класу, то оператор

obj(99.57, "перевизначення");

перетвориться в такий виклик операторної функції operator():

operator()(99.57, "перевизначення");

У загальному випадку при перевизначенні оператора виклику функцій "()" визначаються параметри, які необхідно передати функції operator(). Під час використання оператора "()" у програмі задані аргументи копіюються в ці параметри.

Як завжди, об'єкт, який здійснює виклик операторної функції (obj у наведеному прикладі), адресується покажчиком this.

Розглянемо приклад перевизначення оператора виклику функцій "()" для класу kooClass. Тут створюється новий об'єкт класу kooClass, координати якого є результатом підсумовування відповідних значень координат об'єкта і значень, що передаються як аргументи.

class kooClass

{ intx, y, z; // тривимірні координати

public:

kooClass() { x = y = z = 0; }

kooClass(int c, int d, int f) {x = c; y = d; z = f; }

kooClass operator()(int a, int b, int c);

void Show(char \*s);

};

kooClass kooClass::operator()(int a, int b, int c) // перевизначення оператора виклику функцій "()"

{ kooClass tmp; // створення тимчасового об'єкта

tmp.x = x + a;

tmp.y = y + b;

tmp.z = z + c;

return tmp; // повертає модифікований тимчасовий об'єкт

}

void kooClass::Show(char \*s) // відображення тривимірних координат x, y, z

{ cout << "Координати об'єкта <" << s << ">: ";

cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;

}

int main()

{ kooClass ObjA(1, 2, 3), ObjB;

ObjB = ObjA(10, 11, 12); // виклик функції operator()

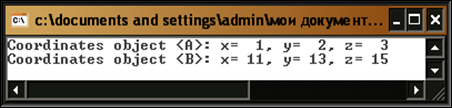
ObjA.Show("A");

ObjB.Show("B");

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:



Не забувайте, що при перевизначенні оператора виклику функцій "()" можна використовувати параметри будь-якого типу, та і сама операторна функція operator() може повертати значення будь-якого типу. Вибір типу повинен диктуватися потребами конкретних програм.

## Перевизначення операторів з використанням функцій-не членів класу

Перевизначення бінарних і унарних операторів для класу можна реалізувати і з використанням функцій, які не є членами класу. Однак такі функції необхідно оголосити "друзями" класу.

Оскільки функції-не члени класу (у тому числі і функції-"друзі") не мають покажчика this, то, якщо для перевизначення бінарного оператора використовується "дружня" функція класу, то для виконання певної операції операторній функції потрібно безпосередньо передати обидва операнди. Якщо ж за допомогою "дружньої" функції класу перевизначається унарний оператор, то операторній функції передається один операнд.

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

=, (), [], –>, new і delete.

### **Перевизначення унарних операторів за допомогою "дружніх" функцій класу**

Перевантаження операторів у вигляді дружніх функцій нічим не відрізняється від перевантаження за допомогою методів класу, за одним виключенням — функція тепер не одержує неявний вказівник \*this. Тому всі операнди повинні бути вказані явно.

При цьому спроба передати операторній "дружній" функції безпосередньо операнд успіху не матиме:

// Цей варіант перевизначення операторної функції працювати не буде

kooClass operator++(kooClass obi)

{ obi.x++;

obi.y++;

obi.z++;

return obi;

}

Причина полягає у тому, що операторній функції operator++() через параметр obi передається лише копія об'єкта, яка активізує виклик даної операторної функції. Таким чином, зміни в тілі функції operator++() не впливають на об'єкт, що викликається, позаяк вони змінюють тільки локальний параметр.

Щоб використати саме "дружню" функцію класу для перевизначення унарних операторів, необхідно передати їй об'єкт за посиланням. Оскільки посилальний параметр є неявним покажчиком на аргумент, то зміни, внесені в параметр, вплинуть і на аргумент.

Перевантаження унарного мінуса. Розглянемо приклад, у якому унарний оператор “-” перевантажується дружньою функцією:

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& obj) { Re = obj.Re; Im = obj.Im; }

~TComplex(){}

friend void print(TComplex z);

friend TComplex operator-(TComplex obj);

};

int main()

{ TComplex z(1,1),u(0,0);

print(z);

u=-z;

print(u);

getch();

}

void print(TComplex obj) { printf("%lf + i\*%lf\n", obj.Re, obj.Im); }

TComplex operator-(TComplex obj)

{ TComplex tmp (0,0);

tmp.Re = -obj.Re;

tmp .Im = -obj.Im;

return tmp;

}

Як бачимо, на відміну від попередніх варіантів, унарний мінус перевантажується операторною функцією, що має один операнд. Отже, у тілі функції необхідно створити локальний об'єкт, а потім змінити його зміст відповідно до визначення оператора.

До недоліку такого підходу варто віднести додаткові витрати пам'яті і часу при створенні локальних об'єктів.

Перевантаження операторів інкремента і декремента. Оскільки дружні функції не одержують вказівник this, операнди операторів ″++″ і ″--″ необхідно передавати за посиланням. Якщо зневажити цим правилом і передати операнд за значенням, то інкрементація і декрементація торкнуться лише копії параметра, а не оригіналу. Можна було б створити локальний об'єкт, скопіювати туди значення параметра, а потім повернути його в модуль, звідки виконується виклик, але це занадто неефективно. Передача параметра за посиланням усе спрощує.

Якщо для перевизначення операторів інкремента або декремента використовується "дружня" функція класу, то її префіксна форма приймає один параметр (який і є операндом), а постфіксна форма – два параметри (другим є цілочисельне значення, яке не використовується).

Наприклад,

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& obj) { Re = obj.Re; Im = obj.Im; }

~TComplex() { }

void print();

friend TComplex& operator++(TComplex& obj);

friend const TComplex operator++(TComplex& obj, int i);

friend TComplex& operator--(TComplex& obj);

friend const TComplex operator--(TComplex& obj, int i);

friend void print(const TComplex& obj);

};

int main()

{ TComplex obj(1,1);

++obj; print(obj);

obj++; print(obj);

--obj; print(obj);

obj--; print(obj);

getch();

}

void print(const TComplex& obj)

{ printf("%lf + i\*%lf\n", obj.Re, obj.Im);

}

TComplex& operator++(TComplex& obj)

{ ++obj.Re;

++obj.Im;

printf("Prefix**-**++ \n");

return obj;

}

const TComplex operator++(TComplex& obj, int i)

{ ++obj.Re;

++obj.Im;

printf("Postfix++ \n");

return obj;

}

TComplex& operator--(TComplex& obj)

{ --obj.Re;

--obj.Im;

printf("Prefix-- \n");

return obj;

}

const TComplex operator--(TComplex& obj, int i)

{ --obj.Re;

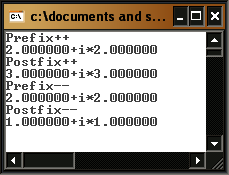
--obj.Im;

printf("Postfix-- \n");

return obj;

}

Результат цієї програми:



### **Використання "дружніх" функцій для перевизначення бінарних операторів**

У багатьох випадках при перевизначенні операторів використання "дружніх" функцій класу немає ніякої переваги порівняно з використанням методів класу. Проте часто трапляються ситуації, у яких "дружня" функція класу виявляється надзвичайно корисною. Наприклад, коли потрібно, щоб зліва від бінарного оператора знаходився об'єкт вбудованого типу. Окрім того, у деяких випадках перевантаження арифметичних операторів за допомогою методів класу може привести до небажаних ефектів.

Розглянемо наступну реалізацію перевантаження оператора “+” за допомогою метода класу:

class TComplex

{ double Re;

double Im;

public:

TComplex(double x, double y):Re(x), Im(y) { }

TComplex(TComplex& obj) { Re = obj.Re; Im = obj.Im; }

~TComplex(){}

void print();

const TComplex operator+(const TComplex& obj);

};

int main()

{ TComplex ObjA(1,1), ObjC(0,0);

ObjC = ObjA + 1.0;

ObjC.print();

getch();

}

const TComplex TComplex::operator+(const TComplex& obj)

{ TComplex tmp(0,0);

tmp.Re = Re+ obj.Re;

tmp.Im = Im+ obj.Im;

return tmp;

}

void TComplex::print()

{ printf("%lf + i\*%lf\n", Re, Im);

}

Оскільки другий параметр конструктора задається за замовчуванням, відкривається можливість неявного перетворення змінної типу double в об'єкт класу TComplex. Отже, вираз

ObjC = ObjA + 1.0;

є цілком коректним (оскільки об'єкт ObjA знаходиться зліва від операції додавання "+", то він викликає операторну функцію, яка (імовірно) здатна виконати операцію додавання дійсного значення з деяким елементом об'єкта ObjA).

Одержуємо результат: 2.000000 + i\*1.000000.

Однак, особливості перевантаження операторів за допомогою методів порушують математичні правила. Наведений нижче вираз є невірним:

1.0 + ObjA.

Це обумовлюється тим, що лівим і правим операндами операторної функції operator+() повинні бути об'єкти класу TComplex. Якщо як правий об'єкт зазначена змінна типу double, вона буде неявно перетворена в об'єкт класу TComplex. На об'єкт же, що викликає функцію operator+(), це не поширюється. Компілятор просто повідомить, що для цієї ситуації в класі немає придатної функції-члена.

Вирішенням такої проблеми є перевизначення оператора додавання "+" з використанням двох "дружніх" функцій класу. У цьому випадку операторній функції безпосередньо передаються обидва операнди, після чого вона виконується подібно до будь-якої іншої перевизначеної функції, тобто на основі типів її аргументів. Одна версія операторної функції operator+() оброблятиме аргументи об'єкт + double -значення, а інша – аргументи double -значення + об'єкт.

Наприклад,

class TComplex

{ double Re, Im;

public:

TComplex(double x, double y) : Re(x), Im(y) { }

void print();

friend TComplex operator+(float x, TComplex y);

friend TComplex operator+(TComplex x, float y);

};

TComplex operator+(TComplex x, float y)

{ TComplex tmp(0,0);

tmp.Re = x.Re+y;

tmp.Im = x.Im+y;

return tmp;

}

TComplex operator+(float x, TComplex y)

{ TComplex tmp(0,0);

tmp.Re = x+y.Re;

tmp.Im = x+y.Im;

return tmp;

}

void print()

{ printf("%lf + i\*%lf \n", Re, Im);

}

int main()

{ TComplex ObjA(1,1), ObjB(0,0);

ObjB = 2.5+ObjA;

ObjB.print();

ObjA =ObjB+3.7;

ObjA.print();

getch();

}

Перевизначення бінарного оператора додавання "+" (або будь-якого іншого бінарного оператора: "– ", "\*", "/") з використанням"дружніх" функцій класу дає змогу розташовувати значення вбудованого типу як справа, так і зліва від операції.

Дружні операторні функції використовують, якщо необхідно, щоб перший аргумент операції не був об'єктом класу. Це потрібно, наприклад, для забезпечення комутативності операцій.