Практична робота №4. МЕХАНІЗМИ ПЕРЕВИЗНАЧЕННЯ ОПЕРАТОРІВ

Зміст

Для покращення роботи з об'єктами класового типу у мові програмування C++ оператори можна перевизначати. Отриманий від цього виграш дає змогу органічно інтегрувати нові типи даних користувача у середовище програмування.

Перевизначаючи оператор, можна встановити певну його дію для конкретного класу. Наприклад, клас, який визначає зв'язний список, може використовувати оператор додавання "+" для внесення об'єкта до списку. Клас, який реалізує стек, може використовувати оператор додавання "+" для запису об'єкта в стек. У будь-якому іншому класі аналогічний оператор міг би слугувати для виконання абсолютно інших операцій. Під час перевизначення оператора жодне з оригінальних значень його об'єктів не втрачають. Перевизначений оператор (у своїй новій якості) працює як абсолютно новий оператор. Наприклад, під час перевизначення бінарного оператора додавання "+" для оброблення зв'язного списку ця функція не призводить до зміни операції додавання стосовно цілочисельних значень.

Механізм перевизначення операторів тісно пов'язаний з механізмом перевизначення функцій. Щоб перевизначити оператор, необхідно визначити дію нової операції для класу, до якого

вона застосовуватиметься. Для цього створюють функцію operator (операторна функція), яка визначає дію цього оператора. Її загальний формат є таким:

```
// Створення операторної функції
тип ім'я_класу::operator#(перелік_аргументів)
{
    Операції над об'єктами класу
}
```

У цьому записі перевизначена операція позначена символом "#", а елемент *тип* вказує на тип значення, що повертається внаслідок виконання даної операції. І хоча він у принципі може бути будь-яким, тип значення часто збігається з іменем класу, для якого перевизначають функцію operator. Такий зв'язок полегшує використання перевизначеного оператора у складних арифметичних і логічних виразах. Як буде показано далі, конкретне значення елемента *перелік_аргументів* визначають декількома чинниками.

4.1. Механізми перевизначення операторів з використанням функцій-членів класу

Операторна функція може бути членом класу або бути поза ним. Операторні функції, які не є членами класу, визначають як його "друзі". Операторні функції-члени і не члени класу відрізняються між собою механізмом перевизначення. Кожний з механізмів перевизначення операторів спробуємо розглянути більш детально на конкретних прикладах.

4.1.2. Перевизначення бінарних операторів додавання "+" і присвоєння "="

Почнемо з простого прикладу. У наведеному нижче коді програми створено клас cooClass, який підтримує дії з координатами об'єкта в тривимірному просторі. Для класу cooClass перевизначають оператори додавання "+" і присвоєння "=". Отже, розглянемо уважно код цієї програми.

Код програми 4.1. Демонстрація механізму перевизначення бінарних операторів додавання "+" та присвоєння "=" за допомогою функцій-членів класу

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class cooClass {                 // Оголошення класового типу
    int x, y, z;                 // Тривимірні координати
public:
    cooClass() {x = y = z = 0; }
    cooClass(int c, int d, int f) {x = c; y = d; z = f; }
    cooClass operator+(cooClass obj);    // Операнд obj передається неявно.
    cooClass operator=(cooClass obj);    // Операнд obj передається неявно.
    void Show(char *s);
};

// Перевизначення бінарного оператора додавання "+".
cooClass cooClass::operator+(cooClass obj)
{
    cooClass tmp;                // Створення тимчасового об'єкта
    tmp.x = x + obj.x;           // Операції додавання цілочисельних значень
    tmp.y = y + obj.y;           // зберігають початковий вміст операндів
    tmp.z = z + obj.z;

    return tmp;                  // Повертає модифікований тимчасовий об'єкт
```

```
}

// Перевизначення оператора присвоєння "=".
cooClass cooClass::operator=(cooClass obj)
{
        x = obj.x;     // Операції присвоєння цілочисельних значень
        y = obj.y;     // зберігають початковий вміст операндів
        z = obj.z;

        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}
// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;
        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA + ObjB;            // Додавання об'єктів ObjA і ObjB і присвоєння об'єкту ObjC
        ObjC.Show("C=A+B");
        ObjC = ObjA + ObjB + ObjC;  // Множинне додавання об'єктів і присвоєння об'єкту ObjC
        ObjC.Show("C=A+B+C");
        ObjC = ObjB = ObjA;            // Множинне присвоєння об'єктів
        ObjB.Show("B=A");
        ObjC.Show("C=B");
        return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A+B>:      x= 11, y= 12, z= 13
Координати об'єкта <C=A+B+C>:    x= 22, y= 24, z= 26
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
```

Аналізуючи код цієї програми, можна побачити, що обидві операторні функції мають тільки по одному параметру, хоча вони перевизначають бінарні операції. Цю, на перший погляд, "кричущу" суперечність можна легко пояснити. Йдеться про те, що під час перевизначення бінарного оператора з використанням функції-члена класу їй передають безпосередньо тільки один аргумент. Другий же опосередковано передається через покажчик this. Отже, у рядку

```
tmp.x = x + obj.x;
```

під членом-даних x маємо на увазі член this->x, тобто член x зв'язується з об'єктом, який викликає дану операторну функцію. В усіх випадках опосередковано передається об'єкт, який вказано зліва від символу операції, тобто той, який став причиною виклику операторної функції. Об'єкт, який розташовано праворуч від символу операції, передано цій функції як аргумент. У загальному випадку під час застосування функції-члена класу для перевизначення унарного оператора параметри не використовують взагалі, а для перевизначення бінарного оператора беруть до уваги тільки один параметр[1]. У будь-якому випадку об'єкт, який викликає операторну функцію, опосередковано передається через покажчик this.

[1] Тернарний оператор "?" перезавантажувати не можна.

Необхідно пам'ятати! Якщо для перевизначення бінарного оператора використовують функцію-члена класу, то об'єкт, який знаходиться зліва від операції, викликає операторну функцію і передається їй опосередковано через покажчик this. Об'єкт, який розташовується праворуч від операції, передається операторній функції як параметр.

Щоб зрозуміти механізм перевизначення операторів, розглянемо уважно наведену вище програму, починаючи з перевизначеного оператора додавання "+". Під час оброблення двох об'єктів типу cooClass оператором додавання "+" виконуються операції додавання значень відповідних координат так, як це показано у функції operator+(). Але зауважте, що ця операторна функція не модифікує значень жодного операнда. Як результат виконання операції ця функція повертає об'єкт типу cooClass, який містить результати попарного додавання координат двох об'єктів. Щоб зрозуміти, чому операція "+" не змінює вміст жодного з об'єктів-учасників, розглянемо стандартну арифметичну операцію додавання, яку застосовують, наприклад, до чисел 10 і 12. Отож, результат виконання операції 10+12 дорівнює 22, але при його отриманні ні число 10, ні 12 не були змінені. Хоча не існує правила, яке б не давало змоги перевизначеному оператору змінювати значення одного з його операндів, все ж таки краще, щоб він не суперечив загальноприйнятим нормам і залишався у згоді зі своїм оригінальним призначенням.

Зверніть увагу на те, що операторна функція operator+() повертає об'єкт типу cooClass, хоча вона могла б повертати значення будь-якого іншого допустимого типу, що визначено мовою програмування C++. Однак, той факт, що вона повертає об'єкт типу cooClass, дає змогу використовувати оператор додавання "+" у таких складних виразах, як ObjA + ObjB + ObjC – множинне додавання. Частина цього виразу (ObjA + ObjB) отримує результат типу cooClass, який потім додається до об'єкта ObjC. І якби ця частина виразу генерувала значення іншого типу (а не типу cooClass), то таке множинне додавання просто не відбулося б.

На відміну від оператора додавання "+", оператор присвоєння "=" модифікує один зі своїх аргументів[1]. Оскільки операторну функцію operator=() викликає об'єкт, який розташований зліва від символу присвоєння "=", то саме цей об'єкт і модифікується внаслідок виконання операції присвоєння. Після виконання цієї операції значення, яке повертає перевизначений оператор, містить об'єкт, який було вказано зліва від символу присвоєння[2]. Наприклад, щоб можна 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.

4.1.2. Перевизначення унарних операторів інкремента "++" та декремента "--"

Можна перевизначати унарні оператори інкремента "++" та декремента "--", або унарні "-" і "+". Як уже зазначалося вище, під час перевизначення унарного оператора за допомогою функції-члена класу операторній функції жоден об'єкт не передається безпосередньо. Операція ж здійснюється над об'єктом, який викликає цю функцію через опосередковано переданий покажчик this. Наприклад, розглянемо дещо змінену версію попереднього коду програми. У наведеному нижче його варіанті для об'єктів типу cooClass визначають бінарну операцію віднімання та унарну операцію інкремента .

Код програми 4.2. Демонстрація механізму перевизначення префіксної форми унарного оператора інкремента "++"

```
#include <iostream>                    // Для потокового введення-виведення
using namespace std;                   // Використання стандартного простору імен
```

[1] Передусім, це становить саму суть присвоєння.
[2] Такий стан речей цілком узгоджується з традиційною дією оператора "=".

```
class cooClass {                                    // Оголошення класового типу
        int x, y, z;                                // Тривимірні координати
public:
        cooClass() {x = y = z = 0; }
        cooClass(int c, int d, int f) {x = c; y = d; z = f; }
        cooClass operator-(cooClass obj);           // Операнд obj передається неявно.
        cooClass operator=(cooClass obj);           // Операнд obj передається неявно.
        cooClass operator++();                      // Префіксна форма оператора інкремента "++"
        void Show(char *s);
};

// Перевизначення бінарного оператора віднімання "-".
cooClass cooClass::operator-(cooClass obj)
{
        cooClass tmp;                               // Створення тимчасового об'єкта
        tmp.x = x - obj.x;                          // Операції віднімання цілочисельних значень
        tmp.y = y - obj.y;                          // зберігають початковий вміст операндів.
        tmp.z = z - obj.z;

        return tmp;                                 // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
cooClass cooClass::operator=(cooClass obj)
{
        x = obj.x; // Операції присвоєння цілочисельних значень
        y = obj.y; // зберігають початковий вміст операндів.
        z = obj.z;
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}
// Перевизначення префіксної форми унарного оператора інкремента "++".
cooClass cooClass::operator++()
{
        x++; // Інкремент координат x, y і z
        y++;
        z++;

        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA - ObjB;                 // Віднімання об'єктів ObjA і ObjB і присвоєння об'єкту ObjC
        ObjC.Show("C = A-B");
```

```
    ObjC = ObjA - ObjB - ObjC;          // Множинне віднімання об'єктів
    ObjC.Show("C = A-B-C");

    ObjC = ObjB = ObjA;                 // Множинне присвоєння об'єктів
    ObjB.Show("B=A");
    ObjC.Show("C=B");

    ++ObjC;                             // Префіксний інкремент об'єкта ObjC
    ObjC.Show("++C");

    ObjA = ++ObjC;                      // Префіксний інкремент об'єкта ObjC
    ObjC.Show("C");
    ObjA.Show("A = ++C");

    return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A-B>:      x= -9, y= -8, z= -7
Координати об'єкта <C=A-B-C>:    x= 0, y= 0, z= 0
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <++C>:        x= 2, y= 3, z= 4
Координати об'єкта <C>:          x= 3, y= 4, z= 5
Координати об'єкта <A=++C>       x= 3, y= 4, z= 5
```

Як видно з останнього рядка результату виконання програми, операторна функція operator++() інкрементує кожну координату об'єкта і повертає модифіковане його значення, яке повністю узгоджується з традиційною дією оператора інкремента "++".

Як уже зазначалося вище, оператори інкремента "++" та декремента "--" мають дві форми – префіксну і постфіксну. Наприклад, оператор інкремента можна використовувати як у префіксній формі

```
++ObjC;
```

так і у постфіксній формі

```
ObjC++;.
```

Як зазначено в коментарях до попереднього коду програми, операторна функція operator++() визначає префіксну форму операції інкремента "++" для класу cooClass. Але це не заважає перевизначати і його постфіксну форму. Оголошення прототипу постфіксної форми унарного оператора інкремента "++" для класу cooClass має такий вигляд:

```
cooClass cooClass::operator++(int notused);
```

Параметр notused не використовує сама функція. Він слугує індикатором для компілятора, який дає змогу відрізнити префіксну форму оператора інкремента від постфіксної[1]. Нижче наведено один з можливих способів реалізації постфіксної форми унарного оператора інкремента "++" для класу cooClass:

```
// Перевизначення постфіксної форми унарного оператора інкремента "++".
cooClass cooClass::operator++(int notused)
{
    cooClass tmp = *this;               // Збереження початкового значення об'єкта
    x++;                                // Інкремент координат x, y і z
    y++;
    z++;
    return tmp;                         // Повернення початкового значення об'єкта
}
```

[1] Цей параметр також використовують як ознака постфіксної форми і для оператора декремента.

Зверніть увагу на те, що ця операторна функція зберігає початкове значення операнда шляхом виконання такої настанови:

```
cooClass tmp = *this;
```

Збережене значення операнда (у об'єкті tmp) повертається за допомогою настанови return. 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, а потім повернути (не забувайте, що постфіксний оператор інкремента не повинен повертати модифіковане значення свого операнда). У наведеному нижче коді програмі реалізовано обидві форми унарного оператора інкремента "++".

Код програми 4.3. Демонстрація механізму перевизначення унарного оператора інкремента "++" з використанням його префіксної та постфіксної форм

```
#include <iostream>                     // Для потокового введення-виведення
using namespace std;                    // Використання стандартного простору імен

class cooClass {                        // Оголошення класового типу
        int x, y, z;                    // Тривимірні координати
public:
        cooClass() { x = y = z = 0; }
        cooClass(int c, int d, int f) {x = c; y = d; z = f; }
        cooClass operator*(cooClass obj);       // Операнд obj передається неявно.
        cooClass operator=(cooClass obj);       // Операнд obj передається неявно.
        cooClass operator++();                  // Префіксна форма оператора інкремента "++"

        // Постфіксна форма оператора інкремента "++"
        cooClass operator++(int notused);

        // Префіксна форма унарного оператора зміни знаку "-"
        cooClass operator-();
        void Show(char *s);
};

// Перевизначення бінарного оператора множення "*".
cooClass cooClass::operator*(cooClass obj)
{
        cooClass tmp;                   // Створення тимчасового об'єкта

        tmp.x = x * obj.x;              // Операції множення цілочисельних значень
        tmp.y = y * obj.y;              // зберігають початковий вміст операндів
        tmp.z = z * obj.z;

        return tmp;                     // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
cooClass cooClass::operator=(cooClass obj)
{
        x = obj.x; // Операції присвоєння цілочисельних значень
        y = obj.y; // зберігають початковий вміст операндів
        z = obj.z;

        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

// Перевизначення префіксної форми унарного оператора інкремента "++".
cooClass cooClass::operator++()
{
```

```
        x++; // Інкремент координат x, y і z
        y++;
        z++;

        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

// Перевизначення постфіксної форми унарного оператора інкремента "++".
cooClass cooClass::operator++(int notused)
{
        cooClass tmp = *this; // Збереження початкового значення об'єкта

        x++;                                    // Інкремент координат x, y і z
        y++;
        z++;
        return tmp;                             // Повернення початкового значення об'єкта
}

// Перевизначення префіксної форми унарного оператора зміни знаку "-".
cooClass cooClass::operator-()
{
        x=-x; // Зміна знаку координат x, y і z
        y=-y;
        z=-z;
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}


// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA * ObjB;             // Множення об'єктів ObjA і ObjB і присвоєння об'єкту ObjC
        ObjC.Show("C=A*B");

        ObjC = ObjA * ObjB * ObjC;      // Множинне множення об'єктів
        ObjC.Show("C=A*B*C");
        ObjC = ObjB = ObjA;             // Множинне присвоєння об'єктів
        ObjC.Show("C=B");
        ObjB.Show("B=A");

        ++ObjC;                         // Префіксна форма операції інкремента
        ObjC.Show("++C");

        ObjC++;                         // Постфіксна форма операції інкремента
        ObjC.Show("C++");

        ObjA = ++ObjC; // Об'єкт ObjA набуває значення об'єкта ObjC після його інкрементування.
```

```
    ObjA.Show("A = ++C");       // Тепер об'єкти ObjA і ObjC мають однакові значення.
    ObjC.Show("C");

    ObjA = ObjC++; // Об'єкт ObjA набуває значення об'єкта ObjC до його інкрементування.
    ObjA.Show("A=C++");         // Тепер об'єкти ObjA і ObjC мають різні значення.
    ObjC.Show("C");

    -ObjC;                      // Префіксна форма операції зміни знаку
    ObjC.Show("-C");

    return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A*B>:      x= 10, y= 20, z= 30
Координати об'єкта <C=A*B*C>:    x= 100, y= 400, z= 900
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <++C>:        x= 2, y= 3, z= 4
Координати об'єкта <C++>:        x= 3, y= 4, z= 5
Координати об'єкта <A=++C>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 4, y= 5, z= 6
Координати об'єкта <A=C++>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 5, y= 6, z= 7
Координати об'єкта <-C>:         x= -5, y= -6, z= -7
```

Як підтверджують останні рядки результату виконання програми, при префіксному інкрементуванні об'єкта ObjC його значення збільшується до виконання операції присвоєння об'єкту ObjA, при постфіксному інкрементуванні – після виконання операції присвоєння.

Необхідно пам'ятати! Якщо символ "++" знаходиться перед операндом, то викликається операторна функція operator++(), а якщо після операнда – то операторна функція operator++(int notused). Аналогічний підхід використовують і для перевизначення префіксної та постфіксної форм оператора декремента для будь-якого класу[1].

Варто знати! Ранні версії мови C++ не містили відмінностей між префіксною і постфіксною формами операторів інкремента і декремента. Раніше в обох випадках викликалася префіксна форма операторної функції. Це потрібно мати на увазі тоді, коли доведеться працювати із старими C++-програмами.

4.1.3. Особливості реалізації механізму перевизначення операторів

Дія перевизначеного оператора стосовно класу, для якого вона визначають, не обов'язково повинна співпадати з стандартними діями цього оператора стосовно вбудованих C++-типів. Наприклад, оператори "<<" і ">>", які вживаються до об'єктів cout і cin, мають мало спільного з аналогічними операторами, що застосовуються у логічних операторах для порівняння значень цілочисельного типу. Але для поліпшення структурованості та читабельності коду програми створюваний програмістом перевизначений оператор повинен за змогою відображати традиційне призначення тої або іншої операції. Наприклад, оператор додавання "+", перевизначений для класу cooClass, концептуально подібний до операції "+", визначеної для цілочисельних типів. Адже безглуздо у визначенні, наприклад, операції множення "*", яка за своєю дією більше нагадуватиме операцію ділення "/". Отже, основна ідея створення програмістом перевизначених операторів – наділити їх новими (потрібними для нього) можливостями, які, зазвичай, пов'язані з їх первинним призначенням.

На перевизначення операторів накладається ряд обмежень. По-перше, не можна змінювати пріоритет операцій. По-друге, не можна змінювати кількість операндів, які приймаються

[1] Для домашньої вправи спробуйте визначити операторну функцію декремента для класу `kooClass`

оператором, хоча операторна функція могла б ігнорувати будь-який операнд. Окрім цього, за винятком оператора виклику функції (про нього піде мова попереду), операторні функції не можуть мати аргументів за замовчуванням. Нарешті, деякі оператори взагалі не можна перевизначати:

:: .* ?

Оператор ".*" – це оператор спеціального призначення, який буде розглянуто нижче.

Значення порядку слідування операндів. Перевизначаючи бінарні оператори, потрібно пам'ятати, що у багатьох випадках порядок слідування операндів має значення. Наприклад, вираз A + B комутативний, а вираз A – B – ні[1]. Отже, реалізуючи перевизначені версії не комутативних операторів, потрібно пам'ятати, який операнд знаходиться зліва від символу операції, а який – праворуч від нього. Наприклад, у наведеному нижче коді програми продемонстровано механізм перевизначення оператора ділення для класу cooClass:

```
// Перевизначення бінарного оператора ділення "/".
cooClass cooClass::operator/(cooClass obj)
{
    cooClass tmp;          // Створення тимчасового об'єкта

    tmp.x = x / obj.x;
    tmp.y = y / obj.y;
    tmp.z = z / obj.z;

    return tmp; // Повертає модифікований тимчасовий об'єкт
}
```

Необхідно пам'ятати! Саме лівий операнд викликає операторну функцію. Правий операнд передається безпосередньо. Ось чому для коректного виконання операції ділення "/" використовують саме такий порядок слідування операндів: x / obj.x.

4.2. Механізми 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. Як уже зазначалося вище, функції-не члени класу (у тому числі і функції-"друзі") не мають покажчика this. Отже, якщо для перевизначення бінарного оператора використовують "дружня" функція класу, то для виконання певної операції операторній функції потрібно безпосередньо передати обидва операнди. Якщо ж за допомогою "дружньої" функції класу перевизначають унарний оператор, то операторній функції передається один операнд. З використанням функцій-не членів класу не можна перевизначати такі оператори:

=, (), [], –>.

Бінарні операторні функції, які не є членами класу, мають два параметри, а унарні (теж не члени класу) – один.

4.2.1. Використання функцій-"друзів" класу для перевизначення бінарних операторів

У наведеному нижче коді програми для перевизначення бінарного оператора додавання "+" використовують "дружня" функція класу.

Код програми 4.4. Демонстрація механізму перевизначення бінарного оператора додавання "+" за допомогою "дружньої" функції класу

```
#include <iostream>                    // Для потокового введення-виведення
```

[1] Іншими словами, A – B не те ж саме, що B – A!

```
#include <conio>            // Для консольного режиму роботи
using namespace std;        // Використання стандартного простору імен

class cooClass {            // Оголошення класового типу
        int x, y, z;        // Тривимірні координати
public:
        cooClass() {x = y = z = 0; }
        cooClass(int c, int d, int f) { x = c; y = d; z = f; }
        friend cooClass operator+(cooClass obi, cooClass obj);
        cooClass operator=(cooClass obj); // Операнд obj передається неявно.
        void Show(char *s);
};

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора додавання "+".
cooClass operator+(cooClass obi, cooClass obj)
{
        cooClass tmp;       // Створення тимчасового об'єкта
        tmp.x = obi.x + obj.x;
        tmp.y = obi.y + obj.y;
        tmp.z = obi.z + obj.z;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
cooClass cooClass::operator=(cooClass obj)
{
        x = obj.x;
        y = obj.y;
        z = obj.z;
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA + ObjB;             // Додавання об'єктів ObjA і ObjB і присвоєння об'єкту ObjC
        ObjC.Show("C=A+B");

        ObjC = ObjA + ObjB + ObjC;      // Множинне додавання об'єктів і присвоєння об'єкту ObjC
        ObjC.Show("C=A+B+C");

        ObjC = ObjB = ObjA ;            // Множинне присвоєння об'єктів
        ObjC.Show("C=B");
        ObjB.Show("B=A");

        return 0;
```

```
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A+B>:      x= 11, y= 12, z= 13
Координати об'єкта <C=A+B+C>:    x= 22, y= 24, z= 26
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
```

Як бачимо, операторній функції operator+() тепер передаються два операнди. Лівий операнд передається параметру obi, а правий – параметру obj.

У багатьох випадках під час перевизначення операторів за допомогою функцій-"друзів" класу немає ніякої переваги порівняно з використанням функцій-членів класу. Проте часто трапляються ситуації (коли потрібно, щоб зліва від бінарного оператора знаходився об'єкт вбудованого типу), у яких "дружня" функція класу виявляється надзвичайно корисною. Щоб зрозуміти це твердження, розглянемо такий випадок. Як уже зазначалося вище, покажчик на об'єкт, який викликає операторну функцію-члена класу, передається за допомогою ключового слова this. Під час використання бінарного оператора функцію викликає об'єкт, який розташований зліва від нього. І це чудово за умови, що лівий об'єкт визначає задану операцію. Наприклад, якщо у нас є певний об'єкт tmp.obj, для якого визначено операцію додавання з цілим числом, тоді такий запис є цілком допустимим виразом:

```
tmp.obj + 10; // працюватиме
```

Оскільки об'єкт tmp.obj знаходиться зліва від операції додавання "+", то він викликає операторну функцію, яка (імовірно) здатна виконати операцію додавання цілочисельного значення з деяким елементом об'єкта tmp.obj. Але наведений нижче вираз працювати не буде:

```
10 + tmp.obj; // не працюватиме
```

Йдеться про те, що у цьому записі константа, яка розташована зліва від оператора додавання "+", є цілим числом, тобто є значенням вбудованого типу, для якого не визначено жодної операції, операндами якої є ціле число і об'єкт класового типу.

Вирішення такого питання базується на перевизначенні оператора додавання "+" з використанням двох функцій-"друзів" 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* + ***int****-значення,* а інша – аргументи ***int****-значення* + *об'єкт.* Перевизначення бінарного оператора додавання "+" (або будь-якого іншого бінарного оператора: "–", "*", "/") з використанням функцій-"друзів" класу дає змогу розташовувати значення вбудованого типу як справа, так і зліва від операції. Механізм перевизначення такої операторної функції показано у наведеному нижче коді програми.

Код програми 4.5. Демонстрація механізму перевизначення бінарних операторів множення "*" і ділення "/" з використанням функцій-"друзів" класу

```
#include <iostream>       // Для потокового введення-виведення
#include <conio>          // Для консольного режиму роботи
using namespace std;      // Використання стандартного простору імен

class cooClass {          // Оголошення класового типу
      int x, y, z;        // Тривимірні координати
public:
      cooClass() {x = y = z = 0; }
      cooClass(int c, int d, int f) { x = c; y = d; z = f; }
      friend cooClass operator*(cooClass obi, int c);
      friend cooClass operator*(int c, cooClass obi);
      friend cooClass operator/(cooClass obi, int c);
      friend cooClass operator/(int c, cooClass obi);
      void Show(char *s);
```

```
};

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора множення "*".
cooClass operator*(cooClass obi, int c)
{
	cooClass tmp;		// Створення тимчасового об'єкта

	tmp.x = obi.x * c;
	tmp.y = obi.y * c;
	tmp.z = obi.z * c;
	return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора множення "*".
cooClass operator*(int c, cooClass obi)
{
	cooClass tmp;		// Створення тимчасового об'єкта

	tmp.x = c * obi.x;
	tmp.y = c * obi.y;
	tmp.z = c * obi.z;
	return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення бінарного оператора ділення "/".
cooClass operator/(cooClass obi, int c)
{
	cooClass tmp;		// Створення тимчасового об'єкта

	tmp.x = obi.x / c;
	tmp.y = obi.y / c;
	tmp.z = obi.z / c;

	return tmp; // Повертає модифікований тимчасовий об'єкт
}
// Перевизначення бінарного оператора ділення "/".
cooClass operator/(int c, cooClass obi)
{
	cooClass tmp;		// Створення тимчасового об'єкта

	tmp.x = c / obi.x;
	tmp.y = c / obi.y;
	tmp.z = c / obi.z;

	return tmp; // Повертає модифікований тимчасовий об'єкт
}


// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
	cout << "Координати об'єкта <" << s << ">: ";
	cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
```

```
    cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;
    int a = 10, b = 5;

    ObjA.Show("A");
    ObjB.Show("B");

    ObjC = ObjA * a; // Множення об'єкта ObjA на число a
    ObjC.Show("C=A*a");

    ObjC = a * ObjA; // Множення числа a на об'єкт ObjA
    ObjC.Show("C=a*A");

    ObjC = ObjB / b; // Ділення об'єкта ObjB на число b
    ObjC.Show("C=B*b");

    ObjC = a / ObjB; // Ділення числа a на об'єкт ObjB
    ObjC.Show("C=a/B");

    return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A*a>:      x= 10, y= 20, z= 30
Координати об'єкта <C=a*A>:      x= 10, y= 20, z= 30
Координати об'єкта <C=B/b>:      x= 2, y= 2, z= 2
Координати об'єкта <C=a/B>:      x= 1, y= 1, z= 1
```

З наведеного вище бачимо, що операторна функція operator*() перевизначають двічі, забезпечуючи при цьому два можливі випадки участі цілого числа і об'єкта типу cooClass в операції додавання. Аналогічно перевизначають двічі операторна функція operator/().

4.2.2. Використання функцій-"друзів" класу для перевизначення унарних операторів

За допомогою функцій-"друзів" класу можна перевизначати й унарні оператори. Але усвідомлення механізму реалізації такого перевизначення вимагатиме від програміста деяких додаткових зусиль. Спершу подумки повернемося до початкової форми перевизначення унарного оператора інкремента "++", визначеного для класу cooClass і реалізованого у вигляді функції-члена класу. Для зручності проведення аналізу наведемо код цієї операторної функції:

```
// Перевизначення префіксної форми унарного оператора інкремента "++"
cooClass cooClass::operator++()
{
    x++;
    y++;
    z++;
    // Повернення модифікованого об'єкта операнда, адресованого покажчиком
    return *this;
}
```

Як уже зазначалося вище, кожна функція-член класу отримує (як опосередковано переданий) аргумент this, який є покажчиком на об'єкт, який викликав цю функцію. Під час перевизначення унарного оператора за допомогою функції-члена класу аргумент безпосередньо не передаються. Єдиним аргументом, необхідним у цій ситуації, є неявний покажчик на викликуваний об'єкт. Будь-які зміни, що вносяться в члени-даних об'єкта, вплинуть на об'єкт, для якого було викликано цю операторну функцію. Отже, у процесі виконання настанови x++ (у попередній функції) буде інкрементовано член-даних x викликуваного об'єкта.

На відміну від функцій-членів класу, функції-не члени (у тому числі і "друзі") класу не отримують покажчика this і, як наслідок, не мають доступу до об'єкта, для якого вони були викликані. Але, як уже зазначалося вище, операторній "дружній" функції операнд передається безпосередньо. Тому спроба створити операторну "дружню" функцію operator++() у такому вигляді успіху не матиме:

```
// Цей варіант перевизначення операторної функції працювати не буде
cooClass operator++(cooClass obi)
{
        obi.x++;
        obi.y++;
        obi.z++;
        return obi;
}
```

Ця операторна функція не працездатна, оскільки тільки копія об'єкта, яка активізує виклик функції operator++(), передається функції через параметр obi. Отже, зміни в тілі функції operator++() не вплинуть на викликуваний об'єкт, позаяк вони змінюють тільки локальний параметр.

Тим не менше, якщо все ж таки виникає бажання використовувати "дружню" функцію класу для перевизначення операторів інкремента або декремента, то необхідно передати їй об'єкт за посиланням. Оскільки посилальний параметр є неявним покажчиком на аргумент, то зміни, внесені в параметр, вплинуть і на аргумент. Застосування посилального параметра дає змогу функції успішно інкрементувати або декрементувати об'єкт, який використовують як операнд.

Отже, якщо для перевизначення операторів інкремента або декремента використовують "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 використовують).

Нижче наведено повний код програми оброблення тривимірних координат, у якій використовують операторна "дружня" функція класу operator++(). Звернемо тільки увагу на те, що перевизначеними є як префіксна, так і постфіксна форми операторів інкремента.

Код програми 4.6. Демонстрація механізму використання "дружньої" функції класу для перевизначення префіксної та постфіксної форми операторів інкремента

```
#include <iostream>        // Для потокового введення-виведення
#include <conio>           // Для консольного режиму роботи
using namespace std;       // Використання стандартного простору імен

class cooClass {           // Оголошення класового типу
        int x, y, z; // Тривимірні координати
public:
        cooClass() {x = y = z = 0; }
        cooClass(int c, int d, int f) {x = c; y = d; z = f; }
        friend cooClass operator*(cooClass obi, cooClass obj);
        cooClass operator=(cooClass obj);
        // Ці функції для перевизначення оператора інкремента "++"
        // використовують посилальні параметри.
        friend cooClass operator++(cooClass &obi);
        friend cooClass operator++(cooClass &obi, int notused);
        void Show(char *s);
};

// Операторна "дружня" функція класу.
cooClass operator*(cooClass obi, cooClass obj)
{
        cooClass tmp;      // Створення тимчасового об'єкта
        tmp.x = obi.x * obj.x;
        tmp.y = obi.y * obj.y;
```

```
        tmp.z = obi.z * obj.z;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}
// Перевизначення оператора присвоєння "=".
cooClass cooClass::operator=(cooClass obj)
{
        x = obj.x;
        y = obj.y;
        z = obj.z;
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

/* Перевизначення префіксної форми унарного оператора інкремента "++"  з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
cooClass operator++(cooClass &obi)
{
        obi.x++;
        obi.y++;
        obi.z++;
        return obi;
}

/* Перевизначення постфіксної форми унарного оператора інкремента "++" з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
cooClass operator++(cooClass &obi, int notused)
{
        cooClass tmp = obi;
        obi.x++;
        obi.y++;
        obi.z++;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA * ObjB;             // Множення об'єктів ObjA і ObjB і присвоєння об'єкту ObjC
        ObjC.Show("C=A*B");

        ObjC = ObjA * ObjB * ObjC;      // Множинне множення об'єктів
        ObjC.Show("c");
        ObjC = ObjB = ObjA ;            // Множинне присвоєння об'єктів
        ObjC.Show("C=B");
        ObjB.Show("B=A");

        ++ObjC;                         // Префіксна форма операції інкремента
        ObjC.Show("++C");
```

```
        ObjC++;                 // Постфіксна версія інкремента
        ObjC.Show("C++");

        ObjA = ++ObjC;          // Об'єкт ObjA набуває значення об'єкта ObjC після інкрементування.
        ObjA.Show("A = ++C");   // У цьому випадку об'єкти ObjA і ObjC
        ObjC.Show("C");         // мають однакові значення координат.

        ObjA = ObjC++;          // Об'єкт ObjA набуває значення об'єкта ObjC до інкрементування.
        ObjA.Show("A=C++");     // У цьому випадку об'єкти ObjA і ObjC
        ObjC.Show("C");         // мають різні значення координат.

        return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A*B>:      x= 10, y= 20, z= 30
Координати об'єкта <C=A*B*C>:    x= 100, y= 400, z= 900
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <++C>:        x= 2, y= 3, z= 4
Координати об'єкта <C++>:        x= 3, y= 4, z= 5
Координати об'єкта <A=++C>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 4, y= 5, z= 6
Координати об'єкта <A=C++>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 5, y= 6, z= 7
```

Необхідно пам'ятати! Для перевизначення бінарних і унарних операторів потрібно використовувати безпосередньо функції-члени класу. Функції-"друзі" класу використовуються мовою програмування C++ в основному для оброблення спеціальних ситуацій.

4.2.3. Перевизначення операторів відношення та логічних операторів

Оператори відношення (наприклад, "==", "<", ">", "<=", ">=", "!=") і логічні оператори (наприклад, "&&" або "||") також можна перевизначати, причому механізм їх реалізації не представляє жодних труднощів. Як правило, перевизначена операторна функція відношення повертає об'єкт того класу, для якого вона перевизначають. А 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 вже розглянутого вище класу cooClass:

```
// Перевизначення операторної функції дорівнює "=="
bool cooClass::operator==(cooClass 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:

```
cooClass 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 настановою if. Як вправу рекомендуємо самостійно реалізувати й інші оператори відношення та логічні оператори для класу cooClass.

4.3. Механізми перевизначення оператора індексації елементів масиву "[]"

На додаток до традиційних перевизначених операторів мова програмування C++ дає змогу перевизначати і більш "екзотичні", наприклад, оператор індексації елементів масиву "[]". У мові програмування C++ (з погляду механізму перевизначення) оператор "[]" вважається бінарним. Його можна перевизначати тільки для класу і тільки з використанням функції-члена класу. Ось як виглядає загальний формат операторної функції-члена класу operator[]().

```
тип ім'я_класу::operator[](int індекс)
{
        //...
}
```

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

Оператор індексації елементів масиву "[]" перевизначають як бінарний оператор.

Припустимо, нехай створено об'єкт ObjA, тоді вираз ObjA[3] перетвориться в такий виклик операторної функції operator[]():

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

Іншими словами, значення виразу, що задається в операторі індексації елементів масиву "[]", 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 наведеному нижче коді програми в класі aClass оголошується масив для зберігання трьох int-значень. Його конструктор ініціалізує кожного члена цього масиву. Перевизначена операторна функція operator[]() повертає значення елемента, що задається його параметром.

Код програми 4.8. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]"

```
#include <iostream>        // Для потокового введення-виведення
#include <conio>           // Для консольного режиму роботи
using namespace std;       // Використання стандартного простору імен

const int size = 3;

class aClass {             // Оголошення класового типу
        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 << "Значення елементів масиву <A>:" << endl;
        for(int i=0; i<3; i++)
```

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

    return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
a[2]= 4
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4
```

Ініціалізація масиву aMas за допомогою конструктора (у цій і наступній програмах) здійснюється тільки з ілюстративною метою. У цьому коді програми функція operator[]() спочатку повертає значення 3-го елемента масиву aMas. Отже, вираз ObjA[2] повертає число 4, яке відображається настановою cout. Потім у циклі виводяться усі елементи масиву.

Необхідно пам'ятати! Щоб оператор індексації елементів масиву "[]" можна було використовувати як зліва, так і праворуч від оператора присвоєння, достатньо вказати значення, що повертається операторною функцією operator[](), як посилання.

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, що повертається операторною функцією operator[](), є посиланням. Цю можливість продемонстровано у наведеному нижче коді програми.

Код програми 4.10. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]" як зліва, так і праворуч від оператора присвоєння

```
#include <iostream>        // Для потокового введення-виведення
using namespace std;       // Використання стандартного простору імен

const int size = 3;

class aClass {             // Оголошення класового типу
    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 << "Значення елементів масиву <A>:" << endl;
    for(int i=0; i<3; i++)
        cout << "aMas[" << i << "]= " << ObjA[i] << endl;

    // Оператор "[]" знаходиться зліва від оператора присвоєння "=".
    ObjA[2] = 25;
    cout << endl << "aMas[2]= " << ObjA[2]; // Тепер відображається число 25.

    return 0;
}
```

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

Оскільки операторна функція operator[]() тепер повертає посилання на елемент масиву, що індексується параметром і, то оператор індексації елементів масиву "[]" можна використовувати зліва від оператора присвоєння, що дасть змогу модифікувати будь-який елемент масиву[1].

Одна з наявних переваг перевизначення оператора індексації елементів масиву "[]" полягає у тому, що за допомогою нього ми можемо забезпечити реалізацію безпечної індексації елементів масиву. Як уже зазначалося вище, у мові програмування C++ можливий вихід за межі масиву у процесі виконання програми без відповідного повідомлення (тобто без генерування повідомлення про динамічну помилку). 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 (в основу якої покладений програмний код попередньої) оснащена засобом контролю потрапляння індексу масиву в допустимий інтервал його перебування.

Код програми 4.11. Демонстрація прикладу організації безпечного масиву

```
#include <iostream>        // Для потокового введення-виведення
using namespace std;       // Використання стандартного простору імен

const int size = 3;

class aClass {             // Оголошення класового типу
        int aMas[size];
public:
        aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
        int &operator[](int i);
};

// Забезпечення контролю потрапляння індексу масиву
// в допустимий інтервал його перебування.
int &aClass::operator[](int i)
{
        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;          // Виникає помилка тривалості виконання, оскільки
                               // значення індексу 3 виходить за межі допустимого інтервалу.
        return 0;
}
```

[1] Звичайно ж, його, як і раніше, можна використовувати і праворуч від оператора присвоєння.

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 настанови

```
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.

4.4. Механізми перевизначення оператора виклику функцій "()"

Можливо, найбільш інтригуючим оператором, якого можна перевизначати, є оператор виклику функції "()". Під час його перевизначення створюють не новий спосіб виклику функцій, а операторна функція, якій можна передати довільну кількість параметрів. Почнемо з такого прикладу. Припустимо, що певний клас містить наведене нижче оголошення перевизначеної операторної функції:

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

І якщо у програмі створено об'єкт obj цього класу, то настанова

```
obj(99.57, "перевизначення");
```

перетвориться в такий виклик операторної функції operator():

```
operator()(99.57, "перевизначення");
```

У загальному випадку під час перевизначення оператора виклику функцій "()" визначають 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 cooClass. Тут створено новий об'єкт класу cooClass, координати якого є результатом підсумовування відповідних значень координат об'єкта і значень, що передаються як аргументи.

Код програми 4.12. Демонстрація механізму перевизначення оператора виклику функцій "()"

```
#include <iostream>        // Для потокового введення-виведення
using namespace std;       // Використання стандартного простору імен

class cooClass {           // Оголошення класового типу
        int x, y, z;    // Тривимірні координати
public:
        cooClass() { x = y = z = 0; }
        cooClass(int c, int d, int f) {x = c; y = d; z = f; }
        cooClass operator()(int a, int b, int c);
        void Show(char *s);
};

// Перевизначення оператора виклику функцій "()".
cooClass cooClass::operator()(int a, int b, int c)
{
        cooClass tmp;      // Створення тимчасового об'єкта
        tmp.x = x + a;
```

```
        tmp.y = y + b;
        tmp.z = z + c;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Відображення тривимірних координат x, y, z.
void cooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        cooClass ObjA(1, 2, 3), ObjB;

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

        ObjA.Show("A");
        ObjB.Show("B");
        return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:    x= 1, y= 2, z= 3
Координати об'єкта <B>:    x= 11, y= 13, z= 15
```

Не забувайте, що під час перевизначення 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.

4.5. Механізми перевизначення рядкових операторів

За винятком таких операторів, як new, delete, ->, ->* і "кома", решту C++-оператори можна перевизначати таким самим способом, як це було показано в попередніх прикладах. Перевизначення операторів new і delete вимагає застосування спеціальних методів, повний опис яких наведено в розд. 8 (він присвячений обробленню виняткових ситуацій). Оператори ->, ->* і "кома" – це спеціальні оператори, детальний перегляд яких виходить за рамки цього навчального посібника. Читачі, яких цікавлять інші приклади перевизначення операторів, можуть звернутися до такої книги [27]. У цьому ж підрозділі розглядатимемо механізм перевизначення тільки рядкових операторів.

4.5.1. Конкатенація та присвоєння класу рядків з рядками класу

Завершуючи тему перевизначення операторів, розглянемо приклад, який часто називають квінтесенцією прикладів, присвячених вивченню механізму перевизначення операторів класу рядків. Незважаючи на те, що C++-підхід до рядків (які реалізуються у вигляді символьних масивів, що завершуються нулем, а не як окремий тип) є дуже ефективним і гнучким, проте початківці C++-програмування часто стикаються з недоліком у понятійній ясності реалізації рядків, яка наявна в таких мовах, як BASIC. Звичайно ж, цю ситуацію неважко змінити, оскільки у мові програмування C++ існує можливість визначити клас рядків, який забезпечуватиме їх реалізацію подібно до того, як це зроблено в інших мовах програмування. Правду кажучи, на початкових етапах розвитку мови програмування C++ реалізація класу рядків була забавою для програмістів. І хоча стандарт мови програмування C++ тепер визначає рядковий клас, який описано далі у цьому навчальному посібнику, проте спробуйте самостійно реалізувати простий варіант

такого класу. Цей приклад наочно ілюструє потужність механізму перевизначення операторів класу рядків.

Код програми 4.13. Демонстрація механізму конкатенації та присвоєння класу рядків

```
#include <iostream>                 // Для потокового введення-виведення
using namespace std;                // Використання стандартного простору імен

class strClass {     // Оголошення класового типу
        char string[80];
public:
        strClass(char *str = "") { strcpy(string, str); }
        strClass operator+(strClass obj); // Конкатенація рядків
        strClass operator=(strClass obj); // Присвоєння рядків
        // Виведення рядка
        void Show(char *s) { cout << s << string << endl; }
};
```

Як бачимо, в класі strClass оголошується закритий символьний масив string, призначений для зберігання рядка. У наведеному прикладі домовимося, що розмір рядків не перевищуватиме 79 байтів. У реальному ж класі рядків пам'ять для їх зберігання повинна виділятися динамічно, однак це обмеження зараз діяти не буде. Окрім цього, щоби не захаращувати логіку цього прикладу, ми вирішили звільнити цей клас (і його функції-члени) від контролю виходу за межі масиву. Безумовно, в будь-якій справжній реалізації подібного класу повинен бути забезпечений повний контроль за помилками.

Клас strClass має один конструктор, який можна використовувати для ініціалізації масиву string з використанням заданого значення або для присвоєння йому порожнього рядка у разі відсутності ініціалізації. У цьому класі також оголошують два перевизначені оператори, які виконують операції конкатенації та присвоєння. Нарешті, клас strClass містить функцію Show(), яка виводить рядок на екран. Ось як виглядають коди операторних функцій operator+() і operator=():

```
// Конкатенація двох рядків
strClass strClass::operator+(strClass obj)
{
        strClass tmp;           // Створення тимчасового об'єкта
        strcpy(tmp.string, string);
        strcat(tmp.string, obj.string);
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Присвоєння одного рядка іншому
strClass strClass::operator=(strClass obj)
{
        strcpy(string, obj.string);
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}
```

Маючи визначення цих операторних функцій, продемонструємо, як їх можна використовувати на прикладі наведеної нижче основної функції main():

```
int main()
{
        strClass ObjA("Всім "), ObjB("привіт"), ObjC;
        ObjA.Show("A: ");
        ObjB.Show("B: ");

        ObjC = ObjA + ObjB;
```

```
        ObjC.Show("C=A+B: ");
        return 0;
}
```

Спочатку вона конкатенує рядки (об'єкти класу strClass) ObjA і ObjB, а потім присвоює результат конкатенації рядку ObjC.

Внаслідок виконання ця програма відображає на екрані такі результати:

```
A: Привіт
B: усім
C=A+B: Привіт усім
```

4.5.2. Конкатенація та присвоєння класу рядків з рядками, завершені нульовим символом

Потрібно мати на увазі, що оператори присвоєння "=" і конкатенації "+" визначено тільки для об'єктів типу strClass. Наприклад, наведена нижче настанова не працездатна, оскільки вона є спробою присвоїти об'єкту ObjA рядок, який завершується нульовим символом:

```
ObjA = "Цього поки що робити не можна.";
```

Але клас strClass, як буде показано далі, можна удосконалити і дати йому змогу виконувати такі настанови.

Для розширення переліку операцій, підтримуваних класом strClass(наприклад, щоб можна було об'єктам типу strClass присвоювати рядки з завершальним нуль-символом, або конкатенувати рядок, який завершується нульовим символом, з об'єктом типу strClass), необхідно перевизначити оператори "=" і "+" ще раз. Спочатку змінимо оголошення класу:

```
// Перевизначення рядкового класу: остаточний варіант
class strClass {     // Оголошення класового типу
        char string[80];
public:
class strClass {     // Оголошення класового типу
        char string[80];
public:
        strClass(char *str = "") { strcpy(string, str); }
        // Конкатенація об'єктів типу strClass
        strClass operator+(strClass obj);
        // Конкатенація об'єкта з рядком, що завершується нулем
        strClass operator+(char *str);
        // Присвоєння одного об'єкта типу strClass іншому
        strClass operator=(strClass obj);
        // Присвоєння рядка, що завершується нулем, об'єкту типу strClass
        strClass operator=(char *str);
        void Show(char *s) { cout << s << string << endl; }
};
```

Потім реалізуємо перевизначення операторних функцій operator+() і operator=():

```
// Присвоєння рядка об'єкту типу strClass, що завершується нулем
strClass strClass::operator=(char *str)
{
        strClass tmp;           // Створення тимчасового об'єкта

        strcpy(string, str);
        strcpy(tmp.string, string);

        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Конкатенація рядка з об'єктом типу strClass, що завершується нулем
```

```
strClass strClass::operator+(char *str)
{
        strClass tmp;           // Створення тимчасового об'єкта

        strcpy(tmp.string, string);
        strcat(tmp.string, str);

        return tmp; // Повертає модифікований тимчасовий об'єкт
}
```

Уважно проаналізуйте коди цих функцій. Зверніть увагу на те, що правий аргумент є не об'єктом типу strClass, а покажчиком на символьний масив, який завершується нулем, тобто звичайним C++-рядком. Але обидві ці функції повертають об'єкт типу strClass. І хоча теоретично вони могли б повертати об'єкт будь-якого іншого типу, весь сенс їх існування і полягає у тому, щоб повертати об'єкт типу strClass, оскільки результати цих операцій приймаються також об'єктами типу strClass. Перевага визначення рядкової операції, у якій як правий операнд бере участь рядок, який завершується нульовим символом, полягає у тому, що воно дає змогу писати деякі настанови в природній формі. Наприклад, наведені нижче настанови є цілком законними:

```
strClass a, b, c;
a = "Привіт усім";  // Присвоєння рядка, який завершує нулем, об'єкту
c = a + "Георгій";      // Конкатенація об'єкта з рядком, що завершується нулем
```

Наведений нижче код програми містить додаткові визначення операторів присвоєння "=" і конкатенації "+".

Код програми 4.14. Демонстрація механізму конкатенації та присвоєння класу рядків з рядками, завершені нульовим символом

```
#include <iostream>         // Для потокового введення-виведення
#include <conio>            // Для консольного режиму роботи
using namespace std;        // Використання стандартного простору імен

class strClass {    // Оголошення класового типу
        char string[80];
public:
        strClass(char *str = "") { strcpy(string, str); }
        // Конкатенація об'єктів типу strClass
        strClass operator+(strClass obj);
        // Конкатенація об'єкта з рядком, що завершується нулем
        strClass operator+(char *str);
        // Присвоєння одного об'єкта типу strClass іншому
        strClass operator=(strClass obj);
        // Присвоєння рядка об'єкту типу strClass, що завершується нулем
        strClass operator=(char *str);
        void Show(char *s) { cout << s << string << endl; }
};

strClass strClass::operator+(strClass obj)
{
        strClass tmp;           // Створення тимчасового об'єкта

        strcpy(tmp.string, string);
        strcat(tmp.string, obj.string);

        return tmp; // Повертає модифікований тимчасовий об'єкт
}

strClass strClass::operator=(strClass obj)
{
        strcpy(string, obj.string);
```

```
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

strClass strClass::operator=(char *str)
{
        strClass tmp;           // Створення тимчасового об'єкта

        strcpy(string, str);
        strcpy(tmp.string, string);

        return tmp; // Повертає модифікований тимчасовий об'єкт
}

strClass strClass::operator+(char *str)
{
        strClass tmp;           // Створення тимчасового об'єкта

        strcpy(tmp.string, string);
        strcat(tmp.string, str);
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

int main()
{
        strClass ObjA("Привіт "), ObjB("всім"), ObjC;

        ObjA.Show("A: ");
        ObjB.Show("B: ");

        ObjC = ObjA + ObjB;
        ObjC.Show("C=A+B: ");

        ObjA = "для програмування, тому що";
        ObjA.Show("A: ");

        ObjB = ObjC = "C++ це супер";
        ObjC = ObjC + " " + ObjA + " " + ObjB;
        ObjC.Show("C: ");

        return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
A: Привіт
B: усім
C=A+B: Привіт усім
A: для програмування, тому що
C: C++ це супер для програмування, тому що C++ це супер
```

Перш ніж переходити до наступного розділу, спробуйте переконатися у тому, що до кінця розумієте, як отримано ці результати. Тепер постарайтеся також самостійно визначати ще деякі інші операції над рядками. Наприклад, спробуйте визначити операцію видалення підрядка на основі оператора вилучення "-". Зокрема, якщо рядок об'єкта A містить фразу "Це важкий-важкий тест", а рядок об'єкта B – фразу "важкий", то обчислення виразу A-B дасть у підсумку "Це – тест". У цьому випадку з початкового рядка були видалені всі входження підрядка "важкий". Визначте також "дружню" функцію, яка б давала змогу рядку, що завершується нулем, знаходитися зліва від оператора конкатенації "+". Нарешті, додайте у програму код, який забезпечує контроль за помилками – виходу індексу за межі масиву.

Варто знати! Для створюваних власних класів завжди є сенс експериментувати з перевизначенням операторів. Як показують приклади цього розділу, механізм перевизначення операторів можна використовувати для залучення нових типів даних у середовище програмування. Це один з найпотужніших засобів мови програмування С++.

4.6. Особливості виконання роботи

Розглянемо ще одну конструкцію класу, в якій використовують перевантаження операторів і дружні інструментальні засоби, це клас векторів. На прикладі цього класу будуть показані подальші особливості розроблення класів, такі як впровадження в об'єкт двох різних способів опису однієї і тієї ж функціональної можливості. Навіть якщо вам не доводиться стикатися з векторами у своїй роботі, багато з нових методів ви зможете застосовувати для інших потреб.

Вектор в інженерній справі та у фізиці це величина, що має значення (модуль) і напрям. Наприклад, якщо ви штовхнете який-небудь предмет, то результат цієї дії залежатиме від того, наскільки сильним був поштовх (величина модуля) і в якому напрямі він був виконаний. Якщо, наприклад, штовхнути падаючу вазу в одному напрямі, то можна запобігти її падінню, тоді як поштовх у протилежному напрямі пришвидшить її сумний кінець. Щоб якомога точніше описати пересування автомобіля, потрібно вказати швидкість (величину модуля) і напрям. Ваші виправдання в конфлікті з дорожнім патрулем, які зводяться до того, що ви не перевищували допустимої швидкості, не переконають його у вашій правоті, якщо ви порушили правила дорожнього руху. Фахівці у області імунобіології та комп'ютерної техніки вкладають в поняття вектора дещо інший сенс, проте поки що ми не зупинятимемося на цьому (версія цього поняття, яку використовують в комп'ютерній техніці, розглядають в наступній практичній роботі). Нижче наведено теоретичні викладки, в яких містяться деякі відомості про вектори, проте для розуміння всіх особливостей мови С++, які проілюстровано в подальших прикладах, в усесторонньому вивченні векторів немає потреби.

4.6.1. Основні відомості про вектори

Багато фізичних величин характеризуються модулем і напрямом. Результат поштовху, наприклад, залежить як від його сили, так і від напряму. Для переміщення об'єкта на екрані комп'ютера також потрібно знати відстань і напрям. Подібного роду явища можна описувати за допомогою векторів. Наприклад, можна описати рух (переміщення) об'єкта на екрані за допомогою вектора, поданого у вигляді стрілки, проведеної з початкової позиції в потрібну точку. Довжина вектора представляє його модуль (тобто як далеко була переміщена точка), а орієнтація стрілки – напрям переміщення (рис. 4.1). Вектор, який показує подібну зміну положення точки, називають *вектором зсуву.*

Рис. 4.1. Опис переміщення за допомогою вектора

Додавання двох векторів має просту геометричну інтерпретацію. Спочатку проведіть один вектор. Потім проведіть другий вектор, що має початок з кінця стрілки першого вектора. І, нарешті, накресліть вектор з початкової точки першого вектора в кінцеву точку другого вектора. Цей третій вектор є сумою перших двох (рис. 4.2). Звернемо увагу на те, що довжина сумарного вектора менше суми довжин векторів-доданків.

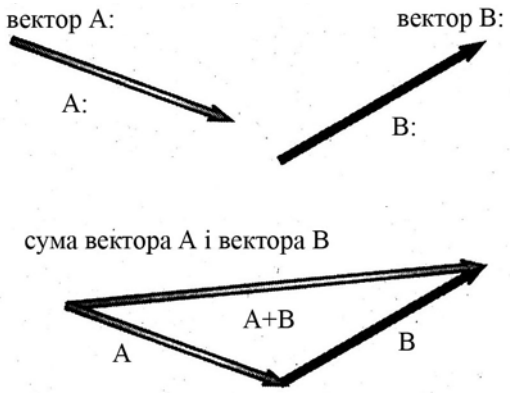

Рис. 4.2. Додавання двох векторів

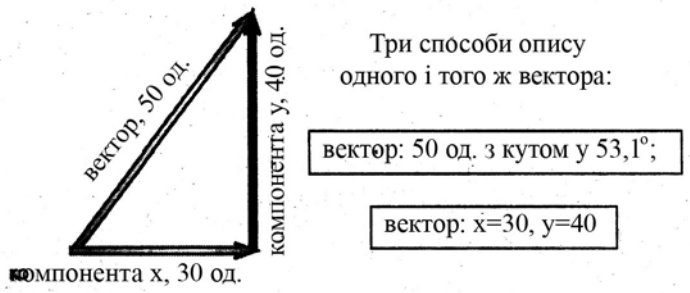

Рис. 4.3. Компоненти *x* і *y* вектора

Вектори добре придатні для ілюстрації перевантаження операторів. По-перше, вектор не можна представити тільки одним числом, тому є сенс створити клас, що представляє вектори. По-друге, у векторному численні існують аналоги звичайних арифметичних операторів, таких як додавання і віднімання. Отже, є можливість здійснити перевантаження відповідних операторів, так щоб можна було застосувати перевантажені операції до векторів.

Для простоти розглядатимемо двовимірні вектори, що виконують переміщення екраном, а не тривимірні, які слугують для подання руху вертольота або птаха. Для опису двовимірного вектора достатньо всього тільки двох чисел, але потрібно визначитися з тим, з яким набором з двох чисел ми працюватимемо:

- можна описати вектор за допомогою його модуля (довжини) і напряму (кута);
- можна представити вектор його компонентами x і y.

Ці компоненти представляють горизонтальну (компоненту *x*) і вертикальну (компоненту *y*) складові вектора, сума яких і становить шуканий вектор. Наприклад, можна описати рух як переміщення точки на 30 одиниць вправо і на 40 одиниць вгору (рис. 4.3). При такому переміщенні точка буде поміщена в те ж місце, що і при переміщенні на 50 одиниць під кутом 53,1° до горизонталі. Тому вектор з модулем 50 і з кутом 53,1° еквівалентний вектору, який має горизонтальну компоненту, що дорівнює 30, а вертикальну 40 одиницям. У разі переміщення векторів має значення тільки те, звідки має початок і де завершується це переміщення, а не маршрут, вибраний для переміщення з початкової в кінцеву точку. Вибір подання в основному аналогічний тому, який виконує перетворення прямокутних координат в полярні і навпаки.

У одних випадках зручно користуватися координатною формою, в інших полярною, тому в опис класу ми вбудуємо обидві форми подання векторів. Окрім цього, ми розробимо клас. Отже, що якщо змінити одне подання вектора, об'єкт автоматично змінить інше подання. Здатність вбудовувати в об'єкт подібного роду інтелектуальні можливості є ще однією перевагою класів мови C++. У коді програми 4.1 подано визначення класу. Щоб освіжити в пам'яті читача інформацію про простір імен, оголошення класу у коді програми поміщене усередині простору імен Vector.

Код програми 4.1. Визначення класу vector.h

```
class VectorClass {
        double x; // горизонтальна координата
        double y; // вертикальна координата
        double mag; // довжина вектора
        double ang; // напрям вектора
        char mode; // 'r'= режим прямокутних координат
        // 'p' = режим полярних координат
        // власні методи для встановлення значень
        void SetMag();
        void SetAng();
        void SetX();
        void SetY();
```

```
public:
        VectorClass();
        VectorClass(double n1, double n2, chat form = 'r');
        Void Set(double n1, double n2, char form = 'r');
        ~VectorClass();
        double XVal() const { return x; } // повертає значення x
        double YVal() const { return y; } // повертає значення y
        double MagVal() const { return mag; } // повертає розміри
        double AngVal() const { return ang; } // повертає положення
        void PolarMode(); // встановлює режим у 'p'
        void RectMode(); // встановлює режим у 'r'

        // перевантаження операторів
        VectorClass operator+(const VectorClass & b) const;
        VectorClass operator-(const VectorClass & b) const;
        VectorClass operator~() const;
        VectorClass operator*(double n) const;

        // дружні функції
        friend VectorClass operator*(double n, const VectorClass & a);
        friend ostrean operator<<(ostream & os, const VectorClass & v);
}; // завершення простору імен Vector
```

Звернемо увагу на те, що чотири функції, які відображають значення компонент, визначені в оголошенні класу. Це автоматично перетворює їх на вбудовані функції. Той факт, що програмний код цих функції такий короткий, робить їх чудовими кандидатами на роль вбудованих функцій. Жодна з них не повинна піддавати змінам дані об'єкта, тому вони були оголошені з використанням специфікатора const. Такі вимоги синтаксису до оголошення функції, що не вносять змін у об'єкт, до якого вона має неявний доступ.

У наведеному коді програми 4.2 показано всі методи і дружні функції, оголошені у коді програми 4.1. У коді програми використана відкрита природа простору імен, щоб додати визначення в простір імен Vector. Звернемо увагу на те, як функції конструктора і функція Set() забезпечують подання конкретного вектора в прямокутній і полярній системах координат. Отже, і той, і інший набір значень негайно опиняться у вашому розпорядженні без подальших обчислень, як тільки в цьому виникає потреба. Окрім цього, математичні функції, вбудовані у мові C++, використовують значення кутів у радіанах, тому у вказаних функціях передбачені перетворення кутів з радіанів в градуси і навпаки у вигляді відповідних методів. Подібна реалізація приховує від користувача такі операції, як перетворення полярних координат в прямокутні або переклад радіан в градуси. Користувачу потрібно знати тільки те, що клас використовує кути в градусах і це забезпечує можливість працювати з векторами в двох еквівалентних поданнях.

Такі проектні рішення відповідають суті ООП, іншими словами, інтерфейс класу зосереджується на найістотнішому (абстрактна модель), тоді як деталі залишаються в тіні. Отже, при роботі з класом VectorClass ви можете думати тільки про основні властивості векторів, таких як їх здатність представляти переміщення і можливість додавання двох векторів. Питання про те, в якій формі представити вектор в записі за допомогою компонент або за допомогою модуля і напряму, відсовується на задній план, оскільки ви можете задати вектор і представити його в тому форматі, який найбільше підходить на даний момент.

Далі розглянемо деякі властивості цього класу детальніше.

Зауваження з сумісності. У деяких системах все ще використовує файл math.h, а не cmath. Окрім цього, деякі системи C++ не виконують автоматичного перегляду бібліотеки математичних функцій.

Код програми 4.2. Програма vector.cpp

```
//vect.cpp методи для класу VectorClass
#include <iostream>
#include <cmath>
```

```
#include "vect.h"
using namespace std;

namespace Vector
{
        const double RadTo_deg=57.2957795130823;
        // власні методи обчислення довжини вектора по x і y
        void VectorClass::SetMag()
        {
            mag = sqrt(x*x + y*y);
        }

        void VectorClass::SetAng()
        {
            if(x == 0.0 && y = 0.00)
                  ang = 0.0;
            else
                  ang = atan2(y, x);
        }

        // встановлення x з полярних координат
        void VectorClass::SetX();
        {
            x = mag*cos(ang);
        }

        // встановлення y з полярних координат
        void VectorClass::SetY();
        {
              y = mag*sin(ang);
        }

        // загальнодоступні методи
        VectorClass::VectorClass() // конструктор за замовчуванням
        {
            x = y = mag = ang =0.0;
            made = 'r';
        }

        // формування вектора з прямокутних координат, якщо стан r
        // (за замовчуванням) або з полярних координат, якщо стан p
        VectorClass::VectorClass(double n1, double n2, char form)
        {
            mode = form;
            if(form == 'r') {
                        x = n1; y = n2;
                        SetMag();
                        SetAng();
            }
            else if(form == 'p') {
                        mag = n1;
                        ang = n2 / RedTo_deg;
                        SetX();
                        SetY();
            }
            else {
                        cout << "Некоректного 3rd параметра до VectorClass() —— ";
                        cout << "векторна множина до 0 " << endl;
                        x = y = mag = ang = 0.0;
            }
        }

        // встановлення вектора з прямокутних координат, якщо стан r
```

```
    // (за замовчуванням) або з полярних координат, якщо стан p
    void VectorClass::Set(double n1, double n2, char form)
    {
        mode = form;
        if(form == 'r') {
                x = n1; y = n2;
                SetMag();
                SetAng();
        else if(fort == 'p') {
                mag = n1;
                ang = n2/RadTo_deg;
                SetX();
                SetY();
        }
        else {
                cout << "Некоректного 3rd параметра до VectorClass() – ";
                cout << "Векторна множина до 0 " << endl;
                x = y = mag = ang = 0.0;
                mode = 'r';
        }
    }

    VectorClass::~VectorClass() // деструктор
    {
    // Знищує об'єкт
    }

    void VectorClass::PolarMode() // встановлення полярного режиму
    {
        mode = 'p';
}

    void VectorClass::RectMode() // встановлення прямокутного режиму
    {
        mode = 'r';
    }

    // перевантаження оператора
    // додавання двох векторів
    VectorClass VectorClass::operator+(const VectorClass & b) const
    {
        return VectorClass(x + b.x, y + b.y);
    }

    // віднімання вектора b з вектора a
    VectorClass VectorClass::operator-(const VectorClass & b) const
    {
        return VectorClass(x - b.x, y - b.y);
    }

    // зміна знаку вектора
    VectorClass VectorClass::operator-() const
    {
        return VectorClass (-x, -y);
    }

    // множення вектора на n
    VectorClass VectorClass::operator*(double n) const
    {
        return VectorClass(n * x, n * y);
    }

    // дружні методи
    // множення n на вектор a
```

```
VectorClass operator*(double n, const VectorClass & a)
{
    return a * n;
}

// відображення прямокутних координат, якщо режим r,
// або відображення полярних координат, якщо режим p
ostream & operator<<(ostream & os, const VectorClass & v)
{
    if(v.mode == 'r')
        os << "(x, y) = (" << v.x << ", " << v.y << ")";
    else if(v.mode == 'p') {
        os << "(m, a) = (" << v.mag << ", " << v.ang*RadTo_deg << ") ";
    }
    else
        os << "Векторний об'єктний режим недійсний";
    return os;
}
} // завершення простору імен Vector


// формування вектора з прямокутних координат, якщо стан r
// (за замовчуванням) або з полярних координат, якщо стан p
VectorClass::VectorClass(double n1, double n2, char form)
{
    mode = form;
    if(form == 'r') {
        x = n1; y = n2;
        SetMag();
        SetAng();
    }
    else if(form == 'p') {
        mag = n1;
        ang = n2 / RedTo_deg;
        SetX();
        SetY();
    }
    else {
        cout << "Некоректного 3rd параметра до VectorClass() —— ";
        cout << "Векторна множина до 0 " << endl;
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}
```

4.6.2. Застосування класу VectorClass для задачі випадкового блукання

У коді 4.3 наведена коротка програма, в якій використовують вдосконалені класи. Вона моделює відоме завдання блукання п'яниці (Drunkard's Walk). У наш час, коли алкоголізм почали вважати зовсім не веселим захворюванням, це завдання звичайно називають завданням випадкового блукання. Ідея полягає в тому, що ви ставите когось у ліхтарного стовпа. Суб'єкт починає ходити туди-сюди, але при цьому на кожному кроці змінює напрям. Одне з формулювань цього завдання може звучати так: скільки кроків буде потрібно суб'єкту, щоб віддалитися від стовпа, наприклад, на 50 футів? У векторній формі це завдання може бути сформульоване так: скільки потрібно скласти випадково орієнтованих векторів, щоб їх сума перевищила 50 футів?

Програма, наведена в коді 4.3, дає змогу задати відстань, яку потрібно пройти в такому режимі, і довжину кроку не зовсім тверезого подорожнього. Вона визначає його поточне положення після кожного кроку (наведене відповідним вектором) і повідомляє, яку кількість кроків буде потрібно, щоб подолати задану відстань, а також кінцеве положення подорожнього (у обох форматах подання). Як ви зможете переконатися самі, досягнення подорожнього не вражають.

Зробивши тисячу кроків у два фути кожен, він віддалився від початкової точки всього тільки на 50 футів. Програма виконує ділення значення абсолютного віддалення від точки старту (50 футів у даному випадку) на кількість кроків, щоб показати, наскільки ефективний цей вид подорожей. Щоб задавати випадкові напрями, програма використовує стандартні функції rand(), srand() і time(), опис яких наведено в наступному розділі. Нагадаємо, що код з програми 4.2 потрібно скомпілювати разом з кодом програми 4.3.

Код програми 4.3. Програма randwalk.cpp

```
// randwalk.cpp використовує клас VectorClass
// компілювати разом з vector.cpp file
#include <iostream>
#include <stdlib> // прототипи функцій rand(), srand()
#include <time> // прототип функції Time()
#include "vect.h"
using namespace std;

int main()
{
        srand(time(0)); // генератор випадкових чисел з початковим значенням
        double direction;
        VectorClass step;
        VectorClass result(0.0, 0.0);
        unsigned long steps = 0;
        double tar, dstep;
        cout << "Введіть tar відстань (q, якщо вихід): ";
        while (cin >> tar)
        {
                cout << "Введіть довжину кроку: ";
                if(!(cin >> dstep)) break;
                while(result.MagVal() < tar)
                {
                        direction = rand() % 360;
                        step.Set(dstep, direction, 'p');
                        result = result + step;
                        steps++;
                }
                cout << "Після " << steps << " кроків, об'єкт має наступне місцеположення: " << endl;
                cout << result << endl;
                result.PolarMode();
                cout << " або" << endl << result << endl;
                cout << "Середня зовнішня відстань за крок = " << result.MagVal()/steps << endl;
                steps = 0;
                result.Set(0.0, 0.0);
                cout << "Введіть tar відстань (q, якщо вихід): ";
        }
        cout << "Бувай!" << endl;
        return 0;
}
```

Результати виконання програми:

```
Введіть tar відстань (q, якщо вихід): 50
Введіть довжину кроку: 2
Після 253 кроків, об'єкт має наступне місцеположення:
(x, y) = (46.1512, 20.4902)
або
(m, a) = (50.495, 23.9402)
Середня зовнішня відстань за крок = 0.199587
Введіть tar відстань (q, якщо вихід): 50
Введіть довжину кроку: 2
Після 951 кроків, об'єкт має наступне місцеположення:
(x, y) = (-21.9577, 45.3019)
або
```

```
(m, a) = (50.3429, 115.8593)
Середня зовнішня відстань за крок = 0.0529362
Введіть tar відстань (q, якщо вихід): 50
Введіть довжину кроку: 1
Після 1716 кроків, об'єкт має наступне місцеположення:
(x, y) = (40.0164, 31.1244)
або
(m, a) = (50.6956, 37.8755)
Середня зовнішня відстань за крок = 0.0295429
Введіть tar відстань (q, якщо вихід): q
Бувай!
```

Цей процес має випадковий характер, що обумовлює значні відмінності між спробами, навіть за незмінних початкових умов. В результаті зменшення кроку наполовину кількість кроків, необхідних для проходження заданої відстані, збільшується в середньому в 4 рази. У теорії ймовірності кількість кроків (N) довжини s, необхідних для віддалення від початкової точки на відстань D, можна задати формулою:

$$N = (D/s)^2$$

Але це всього тільки середнє значення, можливі значні розбіжності між результатами різних спроб. Наприклад, тисяча спроб подолати відстань 50 футів кроками завдовжки по 2 фути дають середнє значення, що становить 636 крокам (достатньо близьке до значення 625, одержаному на підставі теоретичних розрахунків), але при цьому розкид становить від 91 до 3951 кроку. Аналогічно тисяча спроб подолати відстань 50 футів кроками завдовжки по 1 футу дає середнє значення, що становить 2557 крокам (близьке до значення 2500 кроків, одержаному на підставі теоретичних розрахунків) при розкиді від 345 до 10882. Отже, якщо ви потрапите в режим випадкового блукання, не втрачайте присутності духу і дійте за принципом "ширше крок". Ви, зрозуміло, не зможете вибрати потрібний напрям, зате, принаймні, абикуди та прийдете.

4.6.3. Зауваження до програми

Перш за все, звернемо увагу на те, наскільки безболісним було використання простору імен Vector. Using-оголошення

```
using Vector::VectorClass;
```

поміщає ім'я класу VectorClass в діапазон доступу. Оскільки всі методи класу vector мають діапазон доступу класу, імпортування імені класу також робить методи VectorClass доступними без потреби в подальших Using-оголошеннях.

Тепер поговоримо про випадкові числа. Бібліотека із стандарту ANSI, яка також поставляється разом з мовою C++, містить функцію rand(), яка повертає випадкове ціле число в діапазоні від 0 до значення, яке залежить від реалізації. Наша програма використовує операцію обчислення модуля, щоб набути значення кута в діапазоні від 0 до 359. Функція rand() працює за принципом застосування відповідного алгоритму до деякого початкового значення для отримання випадкового числа. Одержане випадкове число слугує початковим значенням при наступному виклику цієї ж функції і т.д. Фактично одержані числа є *псевдовипадковими,* оскільки десять послідовних звернень звичайно дають той самий набір випадкових чисел (точне значення залежить від реалізації). Проте функція srand() надає вам можливість відмовитися від початкового числа, заданого за замовчуванням, і ініціювати іншу послідовність випадкових чисел. Дана програма використовує значення, що повертається функцією time(0) для задавання початкового числа, функція time(0) повертає поточний календарний час, який часто наводять у вигляді кількості секунд, що пройшли з моменту деякої конкретної дати (власне кажучи, функція time() одержує адресу змінної типу time_t, присвоює значення часу цієї змінної і повертає її. Використання значення 0 як адресний аргумент функції дає змогу уникнути використання змінної time_t, даремної у всіх інших відносинах). Отже, оператор

```
srand(time(0));
```

кожного разу, коли ви запускаєте програму, використовує нове початкове значення, завдяки чому число, що випадково згенероване, набуває ще більш випадкового характеру. Заголовний

файл cetdlib (раніше він називався stdlib.h) містить прототипи функцій srand() і rand(), тоді як cTimes (раніше time.h) містить прототип функції time().

В даній програмі для зберігання відомостей про досягнення подорожнього використовує вектор result. На кожній ітерації внутрішнього циклу програма встановлює для вектора step новий напрям і додає його до поточного значення вектора result. Як тільки модуль вектора result перевищить задане значення відстані, цикл уривається.

За рахунок встановлення векторного режиму програма відображає завершальне положення в прямокутних і полярних координатах.

Між іншим, виконання оператора

```
result += step;
```

призводить до того, що подання 'r' встановлюється незалежно від початкових подань векторів result і step. І ось чому. Функція, що реалізовує операцію додавання, створює і повертає новий вектор, що є сумою двох цих аргументів. Ця функція створює такий вектор, використовуючи конструктор, заданий за замовчуванням, а той, водночас, будує вектори в поданні 'r'. Отже, результуючий вектор поданий в режимі 'r'. За замовчуванням присвоєння значень проводиться кожному елементу окремо, тому 'r' присвоюється змінній result.mode. Якщо ви віддаєте перевагу іншій лінії поведінки, наприклад, щоб result зберігав первинне подання, можна, перекрити призначення, задані за замовчуванням, дав відповідне визначення операції присвоєння для даного класу.

4.7. Індивідуальні завдання

Задача 3. Задано кількість пунктів призначення, координати їх місцезнаходження, початок і кінець пунктів проходження і деяка вартість витрат на одиницю проходження маршруту. Потрібно встановити, чи є маршрут, що дає змогу обійти всі пункти призначення при витратах, які не перевищують *k*.

Необхідно розробити програму для визначення маршруту проходження між відповідними пунктами, в якій для зберігання інформації про пункти, потрібно використати масив об'єктів, а для виконання відповідних дій – використати перевантаження операторів. Клас повинен передбачати наявність таких членів-даних:

- назви пунктів призначення (не менше 10 пунктів);
- координати місцезнаходження пунктів призначення;
- витрати на одиницю проходження маршруту;
- задане значення функції мети.

Розроблена програма повинна забезпечити виконання таких дій:

- пункти початку і кінця проходження маршруту;
- отримання інформації про пункти проходження;
- відображення інформації про пункти проходження;
- коригування координат місцезнаходження пунктів;
- коригування витрат на одиницю переходу з одного пункту в інший;
- вибір функції мети розрахунку ($k>0$, min, max);
- виконання розрахунку залежно від вибраної мети;
- обчислення та виведення на екран відповідно до варіанту:
  1) загальної відстані між першим і третім поточними пунктами, починаючи маршрут з другого пункту його призначення, а також загальної вартості проходження маршруту між вказаними пунктами;
  2) загальної відстані між другим і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту його призначення, а також загальної вартості проходження маршруту між вказаними пунктами;
  3) загальної відстані між третім і останнім поточними пунктами, починаючи маршрут з пердостаннього пункту його призначення, а також загальної вартості проходження маршруту між вказаними пунктами;

4) загальної вартості маршруту між другим і четвертим поточними пунктами, починаючи маршрут з третього пункту, а також загальної відстані між вказаними пунктами;
5) загальної вартості маршруту між третім і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту, а також загальної відстані між вказаними пунктами;
6) загальної вартості маршруту між четвертим і передостаннім поточними пунктами, починаючи маршрут з останнього пункту, а також загальної відстані між вказаними пунктами;
7) найкоротшої відстані між пунктами призначення, починаючи маршрут з першого пункту, а також найменшої вартості проходження маршруту між вказаними пунктами;
8) найкоротшої відстані між пунктами призначення, починаючи маршрут з третього пункту, а також найменшої вартості проходження маршруту між вказаними пунктами;
9) найкоротшої відстані між пунктами призначення, починаючи маршрут з останнього пункту, а також найменшої вартості проходження маршруту між вказаними пунктами;
10) найменшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з першого пункту, а також найкоротшої відстані між пунктами призначення;
11) найменшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з третього пункту, а також найкоротшої відстані між пунктами призначення;
12) найменшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з останнього пункту, а також найкоротшої відстані між пунктами призначення;
13) найкоротшої відстані між першим і третім поточними пунктами, починаючи маршрут з другого пункту його призначення, а також найменшої вартості проходження маршруту між вказаними пунктами;
14) найкоротшої відстані між другим і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту його призначення, а також найменшої вартості проходження маршруту між вказаними пунктами;
15) найкоротшої відстані між третім і останнім поточними пунктами, починаючи маршрут з пердостаннього пункту його призначення, а також найменшої вартості проходження маршруту між вказаними пунктами;
16) найменшої вартості маршруту між другим і четвертим поточними пунктами, починаючи маршрут з третього пункту, а також найкоротшої відстані між вказаними пунктами;
17) найменшої вартості маршруту між третім і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту, а також найкоротшої відстані між вказаними пунктами;
18) найменшої вартості маршруту між четвертим і передостаннім поточними пунктами, починаючи маршрут з останнього пункту, а також найкоротшої відстані між вказаними пунктами;
19) найдовшої відстані між пунктами призначення, починаючи маршрут з першого пункту, а також найбільшої вартості проходження маршруту між вказаними пунктами;
20) найдовшої відстані між пунктами призначення, починаючи маршрут з третього пункту, а також найбільшої вартості проходження маршруту між вказаними пунктами;
21) найдовшої відстані між пунктами призначення, починаючи маршрут з останнього пункту, а також найбільшої вартості проходження маршруту між вказаними пунктами;
22) найбільшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з першого пункту, а також найдовшої відстані між пунктами призначення;
23) найбільшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з третього пункту, а також найдовшої відстані між пунктами призначення;
24) найбільшої вартості проходження маршруту між вказаними пунктами, починаючи маршрут з останнього пункту, а також найдовшої відстані між пунктами призначення;
25) найдовшої відстані між першим і третім поточними пунктами, починаючи маршрут з другого пункту його призначення, а також найбільшої вартості проходження маршруту між вказаними пунктами;
26) найдовшої відстані між другим і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту його призначення, а також найбільшої вартості проходження маршруту між вказаними пунктами;
27) найдовшої відстані між третім і останнім поточними пунктами, починаючи маршрут з пердостаннього пункту його призначення, а також найбільшої вартості проходження маршруту між вказаними пунктами;

28) найбільшої вартості маршруту між другим і четвертим поточними пунктами, починаючи маршрут з третього пункту, а також найдовшої відстані між вказаними пунктами;
29) найбільшої вартості маршруту між третім і п'ятим поточними пунктами, починаючи маршрут з четвертого пункту, а також найдовшої відстані між вказаними пунктами;
30) найбільшої вартості маршруту між четвертим і передостаннім поточними пунктами, починаючи маршрут з останнього пункту, а також найдовшої відстані між вказаними пунктами.

Пробні варіанти:

1) загальної відстані між пунктами призначення, починаючи маршрут з першого пункту, а також загальної вартості проходження маршруту між вказаними пунктами;
2) загальної відстані між пунктами призначення, починаючи маршрут з третього пункту, а також загальної вартості проходження маршруту між вказаними пунктами;
3) загальної відстані між пунктами призначення, починаючи маршрут з останнього пункту, а також загальної вартості проходження маршруту між вказаними пунктами;
4) загальної вартості проходження маршруту між вказаними пунктами, починаючи маршрут з першого пункту, а також загальної відстані між пунктами призначення;
5) загальної вартості проходження маршруту між вказаними пунктами, починаючи маршрут з третього пункту, а також загальної відстані між пунктами призначення;
6) загальної вартості проходження маршруту між вказаними пунктами, починаючи маршрут з останнього пункту, а також загальної відстані між пунктами призначення.

Поради що виконання роботи

1. Декартова система координат

1.1. Потрібно взяти аркуш паперу (бажано в клітинку) і по всій його ширині та 5/6 висоти на ньому нанести Декартову систему координат з осями X та Y, бажано тільки І-ий її квадрант. На осях потрібно нанести шкалу в 4 клітинки з відповідними підписами. Рекомендовано по осі X нанести шкалу в 10 од., а по осі Y – 20 од. чи навпаки.

1.2. В межах тільки І-го квадранту потрібно нанести точки (мінімум 10), які будуть вказувати на пункти призначення. Ці точки потрібно пронумерувати і вказати їх координати (x, y), виходячи зі шкал системи координат $0XY$.

1.3. Під Декартовою системою координат потрібно навести таблицю, в якій вказати номери точок та їхні координати.

2. Полярна система координат

2.1. На аркуш паперу, на якому було нанесено Декартову систему координат і точки пунктів призначення, потрібно виконати ще такі додаткові дії: з'єднати кожну точку з точкою перетину координат.

2.2. Визначити кути між віссю $0X$ і кожною прямою, а також довжини кожного з відрізків.

2.3. Отримані значення кута і довжини відрізка потрібно записати у таблицю, наведену під декартовою системою координат.

3. Індивідуальні побудови

3.1. Відповідно до варіанту індивідуального завдання потрібно виконати всі інші побудови, бажано й іншим кольором, а також потрібно нанести відповідні позначення.

3.2. Отримані результати розрахунку потрібно навести на вільному полі паперу.

4.8. Зразок виконання завдання

```
#include <vcl>
#include <iostream>
#include <ctype>
#include <string>
#include <stdlib>
#include <conio>
```