Перевантаження операторів порівняння

Принципи перевантаження операторів порівняння ті ж самі, що і в перевантаженні інших операторів, які ми розглядали на попередніх уроках. Оскільки всі оператори порівняння є бінарними і не змінюють свої ліві операнди, то виконувати перевантаження потрібно через дружні функції.

Наприклад, перевантажимо оператор рівності == і оператор нерівності != для класу Car:

```
#include <iostream>
#include <string>

class Car
{
private:
	std::string m_company;
	std::string m_model;

public:
	Car(std::string company, std::string model)
		: m_company(company), m_model(model)
	{
	}

	friend bool operator== (const Car& c1, const Car& c2);
	friend bool operator!= (const Car& c1, const Car& c2);
};

bool operator== (const Car& c1, const Car& c2)
{
	return (c1.m_company == c2.m_company &&
		c1.m_model == c2.m_model);
}

bool operator!= (const Car& c1, const Car& c2)
{
	return !(c1 == c2);
}

int main()
{
	Car mustang("Ford", "Mustang");
	Car logan("Renault", "Logan");

	if (mustang == logan)
		std::cout << "Mustang and Logan are the same.\n";

	if (mustang != logan)
		std::cout << "Mustang and Logan are not the same.\n";

	return 0;
}
```

Все просто. Оскільки результат виконання оператора != є прямо протилежним результату виконання оператора ==, то ми визначили оператор !=, використовуючи вже перевантажений оператор == (зменшивши, таким чином, кількість коду, складність і можливість виникнення помилок).

А як щодо операторів < і >? Тут потрібно визначитися, чим один об’єкт класу Car може бути кращим за інший об’єкт класу Car, і як це все відобразити в коді. Неочевидно! Тому тут ми і не перевантажували оператори < і >.

Порада: Не перевантажуйте оператори, які є зайвими для вашого класу.

Однак, оператори < і > можна використовувати для сортування списку автомобілів

(об'єктів класу Car) в алфавітному порядку, використовуючи члени m_company і m_model, тому завжди розглядайте різні варіанти.

Деякі класи-контейнери Стандартної бібліотеки C++ вимагають перевантаження оператора <, щоб вони могли зберігати відсортовані елементи.

Перевантажимо оператори порівняння >, <, >= і <=:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    friend bool operator> (const Dollars& d1, const Dollars& d2);
    friend bool operator<= (const Dollars& d1, const Dollars& d2);

    friend bool operator< (const Dollars& d1, const Dollars& d2);
    friend bool operator>= (const Dollars& d1, const Dollars& d2);
};

bool operator> (const Dollars& d1, const Dollars& d2)
{
    return d1.m_dollars > d2.m_dollars;
}

bool operator>= (const Dollars& d1, const Dollars& d2)
{
    return d1.m_dollars >= d2.m_dollars;
}

bool operator< (const Dollars& d1, const Dollars& d2)
{
    return d1.m_dollars < d2.m_dollars;
}

bool operator<= (const Dollars& d1, const Dollars& d2)
{
    return d1.m_dollars <= d2.m_dollars;
}

int main()
{
    Dollars ten(10);
    Dollars seven(7);

    if (ten > seven)
        std::cout << "Ten dollars are greater than seven dollars.\n";
    if (ten >= seven)
        std::cout << "Ten dollars are greater than or equal to seven dollars.\n";
    if (ten < seven)
        std::cout << "Seven dollars are greater than ten dollars.\n";
    if (ten <= seven)
        std::cout << "Seven dollars are greater than or equal to ten dollars.\n";

    return 0;
}
```

Все просто.

Але, як ви вже могли б помітити, оператори > і <= є логічними протилежностями,

тому один з них можна було б визначити через інший. Та ж ситуація і з < і >=. Але, оскільки визначення функцій перевантаження настільки прості, а оператори в рядку оголошення функції так добре поєднуються з операторами в рядку повернення результату, ми вирішили цього не робити.

Перевантаження операторів"<<" і ">>"

Як відомо, оператори "<<" і ">>" у мові C++ перевантажені й дозволяють вводити й виводити дані вбудованих типів. Крім того, оператори "<<" і ">>" можна перевантажити так, щоб вони виводили об'єкти класів, визначених користувачем.

Оператор виведення "<<" називається оператором вставки (insertion operator), тому що він вставляє символи в потік. Аналогічно оператор введення ">>" називається оператором витягу (extraction operator), тому що він витягає символи з потоку. Функції, що перевантажують оператори вставки й витягу, називаються функціями вставки (inserters) і витягу (extractors) відповідно.

Створення власних функцій вставки

Створення власних функцій вставки не викликає особливих труднощів. Усі вони мають такий вигляд.

```
ostream &operator<< (ostream &stream, клас obj)
{
       // Тіло функції вставки
    }
```

Зверніть увагу на те, що функція повертає посилання на потік типу **ostream**. (Нагадаємо, що клас **ostream** є похідним від класу **ios**.) Крім того, перший параметр функції є посиланням на потік виведення. Другий параметр являє собою об'єкт, що підлягає вставці. (Другий параметр може бути також посиланням на цей об'єкт.) Перш ніж закінчити свою роботу, функція вставки повинна повернути покажчик на потік. Це дозволяє використовувати функції вставки усередині складних виразів.

Усередині функції вставки можна виконувати будь-які операції. Властиво, саме вони визначають сутність функції. Однак функцію вставки не слід занадто ускладнювати. Наприклад, зовсім недоцільно змушувати функцію вставки попутно обчислювати число "пі" з точністю до 30 десяткових цифр!

Продемонструємо функцію вставки на прикладі об'єктів класу **phonebook**.

```
class phonebook {
public:
       char name[80];
       int areacode;
       int prefix;
       int num;
       phonebook(char *n, int a, int p, int nm)
       {
              strcpy(name, n);
              areacode = a;
              prefix = p;
              num = nm;
       }
    };
```

У цьому класі зберігаються імена й номера телефонів. Подивимося, як виглядає функція вставки для цього класу.

```
//   Вивести на  екран  ім'я  й номер  телефону.
ostream &operator<< (ostream &stream, phonebook  o) {
       stream << o.name << "    ";
       stream << "(" << o.areacode << ")     ";
       stream << o.prefix << "-" << o.num << "\n";
       return  stream;
    }
```

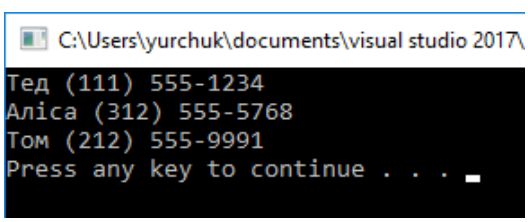
Нижче наведена коротка програма, що ілюструє застосування функції для вставки об'єктів класу **phonebook** у потік виведення

```
#include  <iostream>
#include  <cstring>
#include "windows.h"

using  namespace  std;
class  phonebook {
public:
       char name[80];
       int areacode;
       int  prefix;
       int  num;
       phonebook(char  *n, int  a, int p, int nm)
       {
              strcpy(name, n);
              areacode = a;
              prefix = p;
              num = nm;
       }
};
// Вивести на екран ім'я й номер телефону,
ostream &operator<<(ostream &stream, phonebook o) {
       stream << o.name << " ";
       stream << "(" << o.areacode << ") ";
       stream << o.prefix << "-" << o.num << "\n";
       return stream; // Функція повинна повертати посилання на потік
}
int main() {
SetConsoleCP(1251);
SetConsoleOutputCP(1251);

       phonebook a("Тед", 111, 555, 1234);
       phonebook b("Алiса", 312, 555, 5768);
       phonebook c("Том", 212, 555, 9991);
       cout << a << b << c;
       system("pause");
       return 0;
}
```

Результати роботи цієї програми такі.



Зверніть увагу на те; що функція вставки не є членом класу **phonebook**. На перший

погляд, це видасться дивним, однак причину легко зрозуміти. Якщо операторна функція є членом класу, її лівим операндом (неявно переданим за допомогою покажчика this) є об'єкт, який її викликає. Крім того, цей операнд є об'єктом класу, членом якого є сама операторна функція. Змінити цю ситуацію неможливо. Отже, якщо перевантажена операторна функція є членом класу, її лівий операнд є потоком, а правий - об'єктом даного класу. Отже, перевантажені функції вставки не можуть бути членами класу, для якого вони перевантажуються. Змінні **name**, **areacode**, **prefix** і **num** у попередній програмі є відкритими, тому функція вставки має до них доступ.

Те, що перевантажені функції вставки не можуть бути членами класу, для якого вони визначаються, видається серйозним недоліком. Якщо функція вставки не належить класу, як вона може звертатися до його закритих членів? У попередній програмі всі члени класи були відкритими. Однак інкапсуляція є одним з основних принципів об'єктно-орієнтованого програмування, отже, не можна вимагати, щоб усі члени класу завжди були відкритими. На щастя, ця дилема має розв'язок: необхідно зробити функцію вставки дружньої стосовно зазначеного класу. У цьому випадку перший аргумент перевантаженої функції вставки може залишатися потоком, а сама функція одержить доступ до закритих членів класу, для якого вона визначена. Розглянемо приклад, що ілюструє це приймання.

```
#include  <cstring>
#include "windows.h"
using namespace  std;
class phonebook {
       //  Тепер змінні-  члени закриті
       char name[80];
       int areacode;
       int prefix;
       int num;
public:
       phonebook(char *n, int a, int p, int nm) {
              strcpy(name, n);
              areacode = a;
              prefix = p;
              num = nm;
       }
       friend ostream &operator<<(ostream &stream, phonebook o);
};
// Вивести на екран ім'я й номер телефону,
ostream &operator<<(ostream &stream, phonebook o) {
       stream << o.name << " ";
       stream << "(" << o.areacode << ") ";
       stream << o.prefix << "-" << o.num << "\n";
       return stream; // Функція повинна повертати посилання на потік
}
int main() {
       phonebook a("Ted", 111, 555, 1234);
       phonebook b("Alisa", 312, 555, 5768);
       phonebook c("Tom", 212, 555, 9991);
       cout << a << b << c;
       system("pause");
       return 0;
    }
```

Визначаючи тіло функції вставки, намагайтеся зробити його як можна більш універсальним. Наприклад, функцію вставки з попереднього прикладу можна застосовувати до будь-якого потоку, оскільки вона направляє дані в об'єкт **stream**, що є потоком, що викликають функцію вставки. Можна було б написати оператори

stream << o.name << " ";

або

cout << o.name << " ";

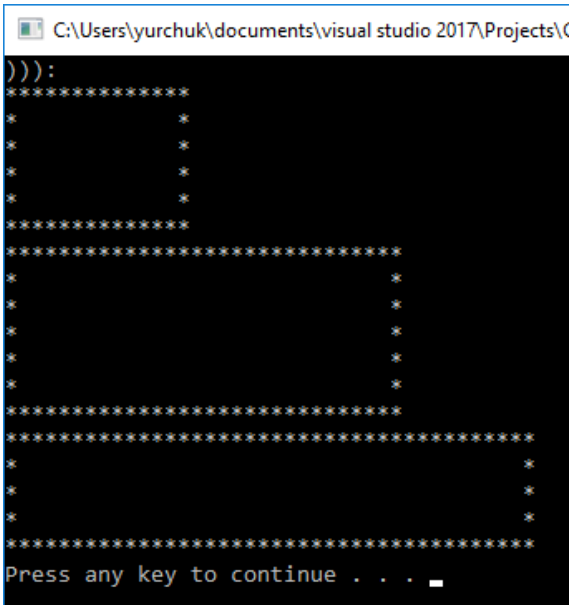
Однак в цьому випадку потік виведення виявився б занадто жорстко заданим. Вихідна версія програми працює з будь-яким потоком, включаючи потік, пов'язаний з файлом. Хоча в деяких ситуаціях, наприклад, при виведенні на спеціальне обладнання в програміста немає вибору, і він може "зашити" потік у програму, у більшості випадків це не так. Чим універсальніша функція вставки, тим вона цінніша.

Перш ніж перейти до функцій витягу, розглянемо ще один приклад функції вставки. Як відомо, функції вставки дозволяють вивести будь-які осмислені дані. Наприклад, така функція для класу, що є частиною системи автоматизованого проектування (CAD - Computer- Aided Design), виводить інструкції плоттера. Інша функція вставки може генерувати графічні зображення. Функції вставки для Windows-додатків можуть виводити на екран діалогові вікна. Щоб продемонструвати виведення об'єктів, які відрізняються від простого тексту, приведемо наступну програму, яка малює прямокутники на екрані. (Оскільки графічні бібліотеки в мові C++ не визначені, програма використовує символи, але їх можна легко замінити графічними зображеннями, якщо конкретна система дозволяє це зробити.)

```
#include  <iostream>

using namespace std;
class box {
       int x, y;
public:
       box(int i, int j) { x = i; y = j; }
       friend ostream &operator<<(ostream &stream, box o); };
              // Виводить прямокутник.
              ostream &operator<< (ostream &stream, box o)
       {
               int i, j;
              for (i = 0; i<o.x; i++) stream << "*";
              stream << "\n";
              for (j = 1; j<o.y - 1; j++) {
                     for (i = 0; i<o.x; i++)
                            if (i == 0 || i == o.x - 1) stream << "*"; else stream << " ";
stream << "\n";
              }
              for (i = 0; i<o.x; i++)
                     stream << "*"; stream << "\n";
              return stream;
       }
       int main() {
              box a(14, 6), b(30, 7), c(40, 5);
              cout << "))):\n"; cout << a << b << c;
              system("pause");
              return 0;
       }
```

Програма виводить на екран наступні рядки.



Створення власних функцій виведення

Функції витягу є антиподами функцій вставки. Вони мають такий вигляд.

```
istream &operator>>(istream &stream, клас &obj)
	{
		//   Тіло функції витягу
		return stream;
	}
```

Функція витягу повертає посилання на потік типу **istream**, тобто на потік уведення. Її першим параметром повинна бути посилання на потік типу **istream**. Зверніть увагу на те, що другим параметром повинна бути посилання на об'єкт класу, для якого перевантажена функція витягу. Це дозволяє модифікувати об'єкт при виконанні операції введення (витягу).

Продовжуючи вдосконалювати клас **phonebook**, напишемо для нього функцію витягу.

```
istream &operator>>(istream &stream, phonebook  &o) {
		cout << "Уведіть  ім'я";
			stream >> o.name;
		cout << "Уведіть  код  міста:    ";
		stream >> o.areacode;
		cout << "Уведіть  префікс:    ";
		stream >> o.prefix;
		cout << "Уведіть  номер:    ";
		stream >> o.num;
		cout << "\n";
		return  stream;
	}
```

Хоча ця функція призначена для введення, при необхідності її можна застосовувати для виведення даних і виконання інших операцій. Однак краще керуватися здоровим глуздом і не використовувати її по прямому призначенню.

Розглянемо приклад використання функції витягу для класу phonebook.

```
#include <iostream>
#include <cstring>
using namespace std;
class phonebook {
	char name[80];
	int areacode;
```

```
        int prefix;
        int num;
public:
        phonebook() { };
        phonebook(char *n, int a, int p, int nm)
        {
                strcpy(name, n);
                areacode = a;
                prefix = p;
                num = nm;
        }
        friend ostream &operator<< (ostream &stream, phonebook o);
        friend istream &operator>>(istream &stream, phonebook &o);
};
// Виводить на екран ім'я й номер телефону,
ostream &operator<<(ostream &stream, phonebook o) {
        stream << o.name << " ";
        stream << "("<< " o.areacode" << ") ";
        stream << o.prefix << "-" << o.num << "\n";
        return stream; // Функція повинна повертати посилання на потік.
}
// Уводить ім'я й номер телефону.
istream &operator>>(istream &stream, phonebook &o)
{
        cout << "Уведіть ім'я: ";
        stream >> o.name;
        cout << "Уведіть код міста: ";
        stream >> o.areacode;
        cout << "Уведіть префікс: ";
        stream >> o.prefix;
        cout << "Уведіть номер:";
                stream >> o.num;
        cout << "\n";
        return stream;
}
int main() {
        phonebook a;
        cin >> a;
        cout << a;
        system("pause");
        return 0;
}
```

Фактично функція витягу для класу **phonebook** не надто ефективна, оскільки оператори cout необхідні, тільки якщо потік уведення пов'язаний з інтерактивним обладнанням, наприклад консоллю (тобто коли потоком уведення є потік **cin**). Наприклад, якщо функція витягу застосовується до потоку, пов'язаного з файлом, оператори **cout** стають непридатними. Забави заради можете спробувати скасувати виконання операторів **cout**, якщо потоком уведення не є потік cin. Наприклад, можна додати в програму оператори виду

```
if (stream == cin)   cout << "Уведіть  ім'я";
```

Тепер запрошення до введення з'явиться тільки в тому випадку, якщо обладнанням уисновку є екран.

Перевантаження операторів за допомогою дружніх функцій

Оператори можна перевантажувати за допомогою дружніх функцій, що не є членами класу. Це значить, що дружні функції не одержують неявного покажчика **this**. Отже, перевантажена операторна функція одержує параметри явно. Таким чином, при

перевантаженні бінарного оператора дружня функція одержує два параметри, а при перевантаженні унарного оператора - один. Першим параметром дружньої функції, що перевантажує бінарний оператор, є його лівий операнд, а другим - правий операнд.

У наступній програмі функція **operator+ ()** є дружньою стосовно класу **loc**.

```
#include <iostream>
using namespace std;
class loc {
	int longitude, latitude;
public:
	loc() {}	// Цей конструктор необхідний для створення
				// тимчасових об'єктів
	loc(int lg, int lt) {
		longitude = lg;
		latitude = lt;
	}
	void show() {
		cout << longitude << " ";
		cout << latitude << "\n";
	}
	friend loc operator+(loc op1, loc op2); // Дружня функція
	loc operator-(loc op2);
	loc operator=(loc op2);
	loc operator++();
};
// Оператор + перевантажується за допомогою дружньої функції.
loc operator+(loc op1, loc op2)
{
	loc temp;
	temp.longitude = op1.longitude + op2.longitude;
	temp.latitude = op1.latitude + op2.latitude;
	return temp;
}
// Перевантажений оператор - для класу loc.
loc loc::operator-(loc op2)
{
	loc temp;
	// Зверніть увагу на порядок операндов
	temp.longitude = longitude - op2.longitude; temp.latitude = latitude - op2.latitude;
	return temp;
}
// Перевантаження оператора присвоювання для класу loc.
loc loc::operator=(loc op2)
{
	longitude = op2.longitude;
latitude = op2.latitude;
return *this; // Вертається об'єкт, що генерує виклик
}
// Перевантажений оператор ++ для класу 1oc.
loc loc::operator++()
{
	longitude++;
	latitude++;
	return *this;
}
int main() {
	loc ob1(10, 20), ob2(5, 30);
	ob1 = ob1 + ob2;
	ob1.show();
	return 0;
    }
```

На застосування дружніх операторних функцій накладають кілька обмежень. По-перше, за допомогою дружніх функцій не можна перевантажувати оператори "=", "()",

"[]" і "->". По-друге, як буде показано в наступному розділі, при перевантаженні операторів інкрементації й декрементаціїи параметр дружньої функції слід передавати по посиланню.

Застосування дружніх функцій для перевантаження операторів"++" і "--"

Якщо дружні функції застосовуються для перевантаження операторів інкрементації й декрементації, їх операнди слід передавати за допомогою посилань. Це необхідно тому, що дружня функція не одержує покажчика **this**. Припустимо, що ми зберігаємо первісний зміст операторів "++" і "--". Отже, оператор "++", наприклад, повинен модифікувати свій операнд. Якщо перевантажити цей оператор за допомогою дружньої функції, як звичайно, то операнд буде передаватися за значенням. Це значить, що дружня функція не зможе змінити свій параметр. Однак цю проблему можна розв'язати, якщо передати операнд дружньої функції за допомогою посилання. У цьому випадку всі зміни усередині функції будуть відбиватися на її фактичному параметрі. Наприклад у наступній програма використовуються дружні функції для перевантаження операторів "++" і "--" у класі **loc**.

```
#include <iostream>
using namespace std;
class loc {
	int longitude, latitude;
public:
	loc() {}
	loc(int lg, int lt) {
		longitude = lg;
		latitude = lt;
	}
	void show() {
		cout << longitude << " ";
		cout << latitude << "\n";
	}
	loc operator=(loc op2);
	friend loc operator++(loc &op);
	friend loc operator--(loc &op);
};
// Перевантажений оператор присвоювання для класу loc.
loc loc::operator= (loc op2)
{
	longitude = op2.longitude;
	latitude = op2.latitude;
	return *this; // Вертається об'єкт, що генерує виклик
}
// Оператор ++ перевантажений дружньою функцією;
// використовується передача параметра за допомогою посилання.
loc operator++(loc &op) {
	op.longitude++;
	op.latitude++;
	return op;
}
// Оператор ++ перевантажений дружньою функцією;
// використовується передача параметра за допомогою посилання.
loc operator-- (loc &op) {
	op.longitude--;
	op.latitude--;
	return op;
}
int main() {
	loc ob1(10, 20), ob2;
	ob1.show();
```

```
    ++ob1;
    ob1.show(); // Виводить на екран числа 11 21
    ob2 = ++ob1;
    ob2.show(); // Виводить на екран числа 12 22
    --ob2;
    ob2.show(); // Виводить на екран числа 11 21
    return 0;
}
```

Якщо необхідно перевантажити постфіксну форму оператора інкрементації або декрементації, слід просто задати другий, фіктивний цілочисельний параметр, як показано нижче.

```
//Перевантаження  постфіксної форми оператора  ++
//   за  допомогою дружньої функції
    friend  loc  operator++(loc  &op, int  x);
```

Дружні операторні функції підвищують гнучкість

У багатьох випадках спосіб перевантаження операторів не важливе: функції-члени й дружні функції еквівалентні. У таких ситуаціях слід віддати перевагу функціям-членам. Однак бувають ситуації, у яких дружні операторні функції дозволяють підвищити гнучкість перевантаженого оператора.

Як відомо, при перевантаженні бінарного оператора за допомогою функції-члена виклик функції здійснюється об'єктом, розташованим у лівій частині оператора. Більше того, функція-член одержує покажчик **this** на цей об'єкт. Тепер допустимо, що в деякому класі операторна функція-член **operator+ ()** додає до об'єкта ціле число. Назвемо об'єкт цього класу іменем **ob**. У такому випадку наступний вираз є цілком припустимим.

```
ob + 100;   //   Усе  правильно
```

Тут об'єкт ob викликає функцію **operator+()**, що виконує додавання. А що відбудеться, якщо цей вираз переписати інакше?

```
100 + ob;   //   Неправильно
```

Тепер у лівій частині оператора знаходиться ціле число. Оскільки воно має вбудований тип, операція додавання цілого числа й об'єкта ob не визначена. Отже, компілятор порахує цей вираз помилковим. Можна собі представити, наскільки ускладниться програмування, якщо ви будете змушені постійно стежити за положенням об'єктів у виразах.

Цю проблему легко розв'язати за допомогою дружньої операторної функції. У цьому випадку дружня функція одержить обидва параметра. Отже, вираз виду об'єкт+ціле число й ціле число+об'єкт стануть правильними, оскільки для дружньої функції не важливий порядок розташування операндів.

Проілюструємо сказане наступною програмою.

```
#include  <iostream>
using namespace std;
class   loc {
       int longitude, latitude;
public: loc() {}
             loc(int lg, int lt) { longitude = lg; latitude = lt; }
```

```
		void show() {
			cout << longitude << " ";
				cout << latitude << "\n";
		}
		friend loc operator+(loc op1, int op2);
		friend loc operator+(int op1, loc op2);
};
// Оператор + перевантажений для додавання об'єкта із цілим числом.
loc operator+(loc op1, int op2)
{
	loc temp;
	temp.longitude = op1.longitude + op2; temp.latitude = op1.latitude + op2;
	return temp;
}
// Оператор + перевантажений для додавання цілого числа з об'єктом,
loc operator+(int op1, loc op2) {
	loc temp;
	temp.longitude = op1 + op2.longitude; temp.latitude = op1 + op2.latitude;
	return temp;
}
int main() {
	loc ob1(10, 20), ob2(5, 30), ob3(7, 14);
	ob1.show(); ob2.show(); ob3.show();
	ob1 = ob2 + 10; // Обоє вирази
	ob3 = 10 + ob2; // вірні
	ob1.show();
	ob3.show();
	return 0;
}
```