Проектування і реалізація програми з перевантаженням функцій

Мета роботи — засвоєння поняття статичного поліморфізму через перевантаження функцій; набуття навичок використання практичних прийомів перевантаження функцій та аргументів за замовчуванням.

Основні завдання роботи

Розробити та реалізувати програмний додаток, який оперує над множенням матриць та векторів. Для цього:

1. Розробити та реалізувати програмно клас *матриця*.
2. Розробити та реалізувати програмно клас *вектор*.
3. Реалізувати програмно перевантаження функції множення вектора на матрицю, матриці на вектор,

матриці на матрицю, множення числа на матрицю, матриці на число.

4. Продемонструвати роботу створених класів та їх функцій в основній частині програми.

Основні теоретичні відомості

1. Поняття поліморфізму

Поліморфізм — це процес, завдяки якому загальний інтерфейс застосовується до двох або більше схожих (але технічно різних) ситуацій, тобто реалізується філософія "один інтерфейс, багато методів".

2. Перевантаження функцій

Після класів важливою можливістю C++ є перевантаження функцій (function overloading). Це той механізм, завдяки якому в C++ досягається один з видів поліморфізму.

У C++ дві або більше функцій є **перевантаженими**, якщо вони мають одне й те саме ім'я, що відрізняються або типом, або кількістю аргументів, або тим і тим.

Щоб перевантажити функцію, треба оголосити та задати всі варіанти, які можуть знадобитися. Компілятор автоматично вибере правильний варіант виклику на основі кількості та/або типу аргументів, що використовуються в функції.

Наведемо приклади перевантаження функцій. В першому з них перевантажені функції відрізняються типом аргументів, у другому — їх кількістю.

Приклад 1. Функція *date( )* перевантажується для отримання дати або у вигляді рядка, або у вигляді трьох цілих чисел. У двох випадках функція виводить на екран передані їй значення:

```
#include <iostream. h>
void date(char *date); // дата у вигляді рядка
void date(int day, int month, int year);
                          // дата у вигляді чисел
main( )
{ data("23/5/95");
  date(23,5,95);
  return 0; }
```

```
//дата у вигляді рядка
void date (char *date)
{  cout <<"Дата:"<<date<<"\n"; }
//дата у вигляді чисел
void date(int day, int month, int year)
{  cout<< "Дата:" << day << "/" << month << "/" <<
          year <<"\n"; }
```

Приклад 2. Перевантажені функції можуть відрізнятися кількістю аргументів:

```
#include <iostream.h>
void f1(int a);
void f1(int a, int b);
main( ) {
    f(10);
    f(10, 20);
    return 0;
}
void f1(int a) {  cout <<"B f1(int a)\n"; }
void f1(int a, int b) { cout<< "Bf1 (int a, int b)\n"; }
```

3. Перевантаження конструкторів

Конструктор перевантажувати можна, деструктор — не можна. Кожному способу оголошення об'єкта класу має відповідати своя версія конструктора класу.

Найчастіше перевантаження конструктора використовується для забезпечення вибору: ініціалізувати об'єкт чи не ініціалізувати. Наведемо приклад:

```
class myclass { int x;
  public:
// перевантаження конструктора двома способами
    myclass( ) {x=0;} //немає ініціалізації
    myclass(int n) {x=n;} //є ініціалізація
    int getx( ) {return x;}
};
```

Перевантаження конструкторів також традиційно застосовується для підтримання масивів. Для співіснування в

програмі неініціалізованих масивів об’єктів поряд з ініціалізованими використовується конструктор, який підтримує ініціалізацію та конструктор, який її не підтримує. Ці ж конструктори можна використовувати для ініціалізації, або неініціалізації об’єктів. Наприклад, для класу *myclass* з попереднього прикладу правильні ці два оголошення:

```
myclass ob(10);
myclass ob[5];
```

4. Конструктор копіювання

Однією з найважливіших форм перевантаженого конструктора є конструктор копій (copy constructor).

Коли об'єкт передається у функцію, виконується порозрядна (тобто точна) копія цього об'єкта та передається тому параметру функції, який отримує об'єкт. Однак бувають ситуації, у яких така точна копія об'єкта небажана. Наприклад, якщо об'єкт містить вказівник на виділену ділянку пам'яті, то в копії вказівник буде посилатися на ту саму ділянку пам'яті, на яку посилається вихідний вказівник. Отже, якщо копія змінює вміст ділянки пам'яті, то ці зміни торкнуться також і вихідного об'єкта. Крім того, коли виконання функції завершується, копія видаляється, що приводить до виклику деструктора цієї копії. Подібна ситуація характерна і для випадку, коли об'єкт є типом значення, яке повертає функція.

Через визначення конструктора копіювання можна повністю контролювати весь процес створення копії об'єкта.

Важливо розуміти, що в C++ точно виділяють два типи ситуацій, у яких значення одного об'єкта передається другому. Перша ситуація — це присвоєння, друга — ініціалізація, яка може виконуватися у трьох випадках:

1) коли в інструкції оголошення об'єкта один об'єкт використовується для ініціалізації другого;

2) коли об'єкт передається у функцію як параметр;

3) коли як значення, що повертається функцією, створюється тимчасовий об'єкт.

Конструктор копій використовується лише для ініціалізації, але не для присвоєння. Після визначення конструктора копій,

він викликається завжди під час ініціалізації одного об'єкта другим.

Основна форма конструктора копій:

```
ім'я_класу (const ім'я_класу &obj) {
  ... // тіло конструктора
}
```

Тут *&obj* — посилання на об'єкт *obj*, що використовується для ініціалізації іншого об'єкта. Наприклад, нехай є клас *myclass*, а *y* —об'єкт типу *myclass*, тоді такі інструкції могли б викликати конструктора копій:

```
myclass x = y; // y явно ініціалізує x
fund (y); // y передається як параметр
y = func2(); // y отримує об’єкт, що повертається
```

У двох перших випадках конструктору копій можна було б передати посилання на об'єкт *y*, у третьому — конструктору копій передається посилання на об'єкт, що повертається функцією *func*().

5. Використання аргументів за замовчуванням

Можливість використання аргументів за замовчуванням (default arguments) пов’язана з перевантаженням функцій та дозволяє при виклику функції відповідний аргумент не задавати, а надати параметру значення за замовчуванням.

Щоб передати параметру аргумент за замовчуванням, треба поставити після параметра знак рівності і те значення, яке треба передати. Тоді, якщо при виклику функції відповідний аргумент не задається, то за замовчуванням функції буде передано задане значення. Наприклад, у наведеній функції двом параметрам за замовчуванням присвоєно значення 0:

```
void f (int a=0, int b=0);
```

Тепер функцію *f( )* можна викликати трьома різними способами: 1) з двома заданими аргументами; 2) тільки з першим заданим аргументом (тоді *b* за замовчуванням

дорівнюватиме нулю); 3) без будь-яких аргументів (тоді *a* та *b* за замовчуванням дорівнюватимуть нулю). Таким чином, всі такі виклики *f( )* правильні:

```
f( ); // a=0 і b=0 за замовчуванням
f(10); // a=10, b=0 за замовчуванням
f(10, 99); // a=10, b=99
```

Якщо створюються функції, які мають один або більше аргументів, що передаються за замовчуванням, ці аргументи потрібно задавати тільки один раз: або у визначенні функції, або в її прототипі.

Усі параметри, що задаються за замовчуванням, мають розміщуватися правіше від параметрів, що передаються звичайним шляхом. Більше того, якщо почали задавати параметри за замовчуванням, то вже не можна передавати параметри звичайним способом.

Аргументи за замовчуванням мають бути або константами, або глобальними параметрами.

Ще два зауваження щодо використання аргументів за замовчуванням:

а) можна передавати аргументи за замовчуванням конструкторам;

б) прикладом використання аргументів за замовчуванням є випадки, коли треба вибрати варіант.

Порядок виконання роботи

1. Для реалізації програмного коду створити два класи: *матриця* і *вектор*. У кожному з класів задати конструктори і деструктори.

2. Як закриті елементи класів створити динамічні масиви, що зберігають відповідно матрицю або вектор та їх розмірність. Динамічний масив матриці має бути двовимірним, вектора — одновимірним.

3. Додати дружні функції множення вектора на матрицю і навпаки для цих двох класів. Як параметри передати об’єкти створених класів.

4. Перевантажити створену функцію для множення вектора на число і навпаки.

5. Перевантажити створену функцію для множення матриці на число і навпаки.

6. Перевантажити створену функцію для множення двох матриць (як об'єктів класу *матриця*).

7. Перевантажити створену функцію для множення двох векторів (як об'єктів класу *вектор*).

8. Під час реалізації перевантажених функцій перевірити можливість такого множення (не всі вектори і матриці можна перемножати).

9. В основній частині програми продемонструвати роботу перевантажених функцій.

Контрольні завдання та запитання

1. Що таке поліморфізм?

2. Які функції називають перевантаженими?

3. Для чого введено перевантаження функцій?

4. Які переваги використання перевантаження конструктора класу?

5. Покажіть, як перевантажити конструктор для наступного класу так, щоб можна було створити неініціалізовані об'єкти. Створюючи неініціалізовані об'єкти, присвойте змінним *x* та *y* значення 0:

```
class myclass { int x,y;
   public:
     myclass(int i, int j) {
       x = i;
       y = j;
     }
   ... // інші функції класу
};
```

6. Поясніть, що таке аргумент за замовчуванням.

7. Що неправильно у наступних прототипах, які використовують аргументи, що передаються за замовчуванням:

```
int f(int count, int max = count);
void func(int x = 4, int y);
```