Обектно ориентирано програмиране

КЛАСОВЕ

ПРИЯТЕЛСКИ КЛАСОВЕ И ФУНКЦИИ. ОПЕРАТОРИ

Приятелски класове и функции

Често е необходимо съвместното използване на два класа. Обектно-ориентираното програмиране налага капсулирането на данните. Достъпът до private-компонентите на даден клас от функция извън класа е забранено. В редица случаи това е сериозно затруднение.

Например, дефинирани са два класа, представящи вектор и матрица съответно. Функцията, която ще реализира произведението на вектор с матрица ще трябва да има достъп до членовете и на двата класа. Един начин за реализирането на това е да се направят член-данните и на двата класа public. Това ще доведе до загубване на предимствата на капсулирането.

Друг начин е да се използват public функции на достъп, осъществяващи достъп до стойностите на член-променливите. Това води до забавяне на изпълнението на програмата.

Приятелски класове и функции

Трети начин за решаване на този проблем е декларирането на функции или класове – приятели на класа. Приятелите на даден клас (функции или класове) имат достъп до всички негови компоненти, т.е. членовете на класа са винаги public за функциите приятели. Ако клас е деклариран като приятел, всички негови член-функции стават функции приятели.

Примери за функции и класове приятели ще разгледаме по-нататък.

Оператори. Предефиниране на оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;
- приоритет;
- асоциативност.

Позицията на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).

Пример: Операторът / е инфиксен (4/8), операторът + е както инфиксен, така и префиксен (2+8, +78), а операторът ++ е както постфиксен, така и префиксен.

Оператори. Предефиниране на оператори ...

Приоритетът определя реда на изпълнение на операторите в операторен терм. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет.

Пример: Приоритетът на * и / е по-висок от този на + и -.

Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има лявоасоциативни и дясноасоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните – отдясно наляво.

В C++ не могат да се дефинират нови оператори, но всеки съществуващ едноаргументен или двуаргументен оператор с изключение на ::, ?:, ., *, # и ## може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.

Оператори. Предефиниране на оператори ...

Например, възможно е да се предефинират операторите +, -, * и /, така че да могат да събират, изваждат, умножават и делят рационални числа. Тогава вместо sum(p, q), sub(p, q), mult(p, q) и quot(p, q) ще можем да пишем p+q, p-q, p*q и p/q, което безспорно е много по-удобно.

Предефинирането се осъществява чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксис като на обикновените функции, но името им се състои от запазената дума **operator**, следвана от мнемоничното означение на предефинирания оператор. Когато предефинирането на оператор изисква достъп до компонентите на класове, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. Предефинираният оператор запазва всички характеристики на оригиналния.

Оператори. Предефиниране на оператори ...

Предефинирането може да стане по два начина:

- чрез функция–приятел;
- чрез член-функция.

Чрез примери ще покажем тези два начина.

Оператори. Предефиниране на оператори ...

Предефиниране чрез функция-приятел

Задача. Да се предефинират операторите +, -, * и / така, че да могат да бъдат използвани за събиране, изваждане, умножение и деление на рационални числа.

В public частта на класа Rational са включени декларациите на предефинираните оператори, предшествани от запазената дума friend:

```
friend Rational operator+(Rational, Rational);
friend Rational operator-(Rational, Rational);
friend Rational operator*(Rational, Rational);
friend Rational operator/(Rational, Rational);
```

а след дефиницията на функцията main са дадени и техните дефиниции.

Оператори. Предефиниране на оператори ...

```
class Rational {
private:
    int numer, denom;
    int gcd(int, int);
public:
    // конструктори
    Rational();
    Rational(int, int);
```

Оператори. Предефиниране на оператори ...

```
    // функции за достъп
    int getNumerator() const;
    int getDenominator() const;
    void print() const;
    // мутатор
    void read();
    friend Rational operator+(Rational, Rational);
    friend Rational operator-(Rational, Rational);
    friend Rational operator*(Rational, Rational);
    friend Rational operator/(Rational, Rational);
};
```

Оператори. Предефиниране на оператори ...

```
…
// предефиниране на оператора +
Rational operator+(Rational p, Rational q) {
   return Rational(p.getNumerator() * q.getDenominator()
                 + p.getDenominator() * q.getNumerator(),
                 p.getDenominator() * q.getDenominator());
}

// предефиниране на оператора -
Rational operator-(Rational p, Rational q) {
   return Rational(p.getNumerator() * q.getDenominator()
                  - p.getDenominator() * q.getNumerator(),
                  p.getDenominator() * q.getDenominator());
}
```

Оператори. Предефиниране на оператори ...

```
// предефиниране на оператора *
Rational operator*(Rational p, Rational q) {
    return Rational(p.getNumerator() * q.getNumerator(),
                    p.getDenominator() * q.getDenominator());
}

// предефиниране на оператора /
Rational operator/(Rational p, Rational q) {
    return Rational(p.getNumerator() * q.getDenominator(),
                    p.getDenominator() * q.getNumerator());
}
```

Оператори. Предефиниране на оператори …

```
…
int main() {
    Rational p(1, 3), q(2, 5), r(p + q);
    r.print();
    cout << endl;
    r = p - q - q;
    r.print();
    cout << endl;
    return 0;
}
```

Оператори. Предефиниране на оператори ...

Забележки:

- Изразът p+q се интерпретира като извикване на операторната функция operator+(p, q).
- Запазва се асоциативността. Изразът p-q-r се интерпретира като (p-q)-r.

Предефиниране чрез член-функция

В този случай първият аргумент на член-функцията трябва да е обект на класа и при дефинирането на операторната функция не се задава като параметър. Ако това не е така, операцията не може да се предефинира като член-функция.

Оператори. Предефиниране на оператори ...

```
class Rational {
private:
   int numer, denom;
   int gcd(int, int);
public:
   // конструктор
   Rational(int = 0, int = 1);
   // функции за достъп
   int getNumerator() const;
   int getDenominator() const;
   void print() const;
```

Оператори. Предефиниране на оператори ...

```
    Rational operator+(Rational) const;
    Rational operator-(Rational) const;
    Rational operator*(Rational) const;
    Rational operator/(Rational) const;
    // мутатор
    void read();
};
```

Оператори. Предефиниране на оператори …

```
…
// предефиниране на оператора +
Rational Rational::operator+(Rational q) const {
   return Rational(getNumerator() * q.getDenominator()
                   + getDenominator() * q.getNumerator(),
                   getDenominator() * q.getDenominator());
}
// предефиниране на оператора -
Rational Rational::operator-(Rational q) const {
   return Rational(getNumerator() * q.getDenominator()
                  - getDenominator() * q.getNumerator(),
                  getDenominator() * q.getDenominator());
}
```

Оператори. Предефиниране на оператори ...

```
// предефиниране на оператора *
Rational Rational::operator*(Rational q) const {
   return Rational(getNumerator() * q.getNumerator(),
                   getDenominator() * q.getDenominator());
}
// предефиниране на оператора /
Rational Rational::operator/(Rational q) const {
   return Rational(getNumerator() * q.getDenominator(),
                   getDenominator() * q.getNumerator());
}
```

Ще отбележим, че в този случай изразът p+q се интерпретира като p.operator+(q).

Оператори. Предефиниране на оператори ...

```
#include <iostream>
#include "Rational.h"
using namespace std;
int main() {
   Rational p(1, 3), q(2, 5), r(p + q);
   r.print();
   cout << endl;
   r = p - q - q;
   r.print();
   cout << endl;
return 0;
}
```

Приложение на средствата за работа с динамичната памет …

Правило на голямата тройка:

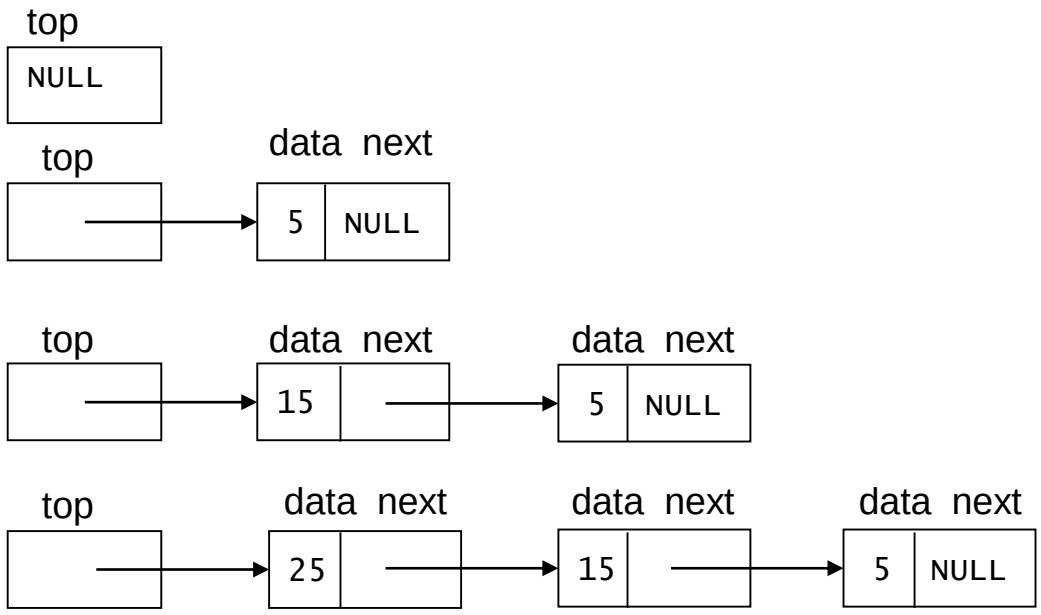
- Деструктор
- Конструктор за копиране
- Оператор за присвояване

Примери:

- Разширяващ се стек
- Свързан стек

Приложение на средствата за работа с динамичната памет

Ще конструираме клас `LinkedStack`, който ще реализира свързаното представяне на стек от цели числа.



Приложение на средствата за работа с динамичната памет ...

Забелязваме, че има указател top, който в първия случай представя празен стек, а в останалите случай – непразен, като сочи двойна кутия с информационна част (data) от тип int и свързваща част (next) от типа на top. Това представяне ще реализираме по следния начин:

```
struct StackElement {
    int data;
    StackElement* next;
};
```

Приложение на средствата за работа с динамичната памет …

След тази дефиниция top представя празен стек. Включването на елемента 5 можем да направим чрез изпълнение на следните действия:

```
StackElement *top = NULL, *p;
p = top;
top = new StackElement;
top->data = 5;
top->next = p;
```

Включването на 15 ще направим по аналогичен начин

```
p = top;
top = new StackElement;
top->data = 15;
top->next = p;
```

Приложение на средствата за работа с динамичната памет …

а на 25 – чрез

```
p = top;
top = new StackElement;
top->data = 25;
top->next = p;
```

Тези разсъждения показват, че който и да е елемент x може да се **включи** в стека чрез изпълнение на фрагмента:

```
p = top;
top = new StackElement;
top->data = x;
top->next = p;
```

Приложение на средствата за работа с динамичната памет …

Изключването на елемент от последния стек води до получаване на стека, илюстриран на по-горната стъпка на същата фигура и може да се реализира така:

```
int x;
p = top;
x = top->data;
top = top->next;
delete p;
```

В x е запомнен изключеният елемент.

Приложение на средствата за работа с динамичната памет …

```
struct StackElement {
   int data;
   StackElement* next;
};

class LinkedStack {
private:
   StackElement* top;
   void copyStack(LinkedStack const&);
   void deleteStack();
public:
   // създаване на празен стек
   LinkedStack();
   // конструктор за копиране
   LinkedStack(LinkedStack const&);

   ~LinkedStack();
```

Приложение на средствата за работа с динамичната памет …

```
    // Оператор =
    LinkedStack& operator=(LinkedStack const &);
    // селектори
    // проверка дали стек е празен
    bool empty() const;
    // намиране на елемента на върха на стека
    int peek() const;
    // мутатори
    // включване на елемент
    void push(int);
    // изключване на елемент
     int pop();
};
```

Статични членове на клас

В C++ може да дефинираме статични членове (член-данни и член-функции) използвайки ключовата дума **static**.

Когато декларираме член-данна на клас като статична, това означава, че независимо колко обекта на класа са създадени, съществува само едно копие на статичната член-данна.

Статичните член-данни се споделят от всички обекти на класа. Памет за тях се заделя в статичната памет.

Статичните член данни могат да се инициализират само извън класа.

Пример: box.cpp

Статични членове на клас

Като се декларира член-функция като статична, това я прави независима от обектите на класа. Статична член-функция може да бъде извикана дори да не съществуват обекти на класа. Те се извикват чрез пълното име

<име на клас>::<име на статична член-функция>(<параметри>);

Имат достъп само до статични член-данни и член-функции.

Указателят **this** не се предава на статичните член функции.

Пример: box2.cpp