Приятелски функции

Досега поставяхме голям акцент на това колко важно е за ООП да се скрива дадена чувствителна информация от нежелан достъп или т.нар. енкапсулация. Това се случваше в C++ благодарение на идентификаторите за достъп.

За да имаме подходящо въведение за това какво представляват приятелските функции е добре да припомним:

- идентификаторите за достъп:
  - `public` - дава достъп на всички външни функции до данните в тази секция
  - `private` - дава достъп единствено на методите на класа до елементите с този идентификатор до данните до данните в тази секция
  - `protected` - дава достъп само до методите на класа, както и до методите на наследниците на класа, не дава достъп на външни функции до данните в тази секция

Но понякога това ограничение ни налага усложнения в писането на код.

Като например (примерът е взет от учебника на Магда Тодорова, има го и в лекциите на доц. Григоров):

Имаме два класа: *вектор* и *матрица*. Искаме да напишем функция, която да реализира произведението на *вектор* и *матрица* и тя съответно трябва да има достъп до член-данните на класовете. Имаме два познати начина да се осигури такъв:

- член-данните на класовете *вектор* и *матрица* да бъдат декларирани с `public` идентификатора, но това ще наруши принципа на енкапсулация на данните
- да се осигури достъп до член-данните по известния ни досега начин - с *getter* и *setter* методи в `public` секцията на класа. Това едновременно и ще работи и ще сме скрили по удачен начин член-данните. Обаче е свързано с извикване на тези методи и малко или много води до едно известно забавяне на програмта, като и до писане на повече и по-сложен код.

Друг пример, който ще използваме е с класа Point, който сме писали на упражнение. По-долу ще видите конкретни имплементации.

Затова сега се запознаваме и с **трети начин** за достъп до член-данните на някой клас, а именно **приятелските функции и класове**.

Те представляват механизъм в C++ за достъп до `private` или `protected` данни.

Името им "*приятелски*" интуитивно подсказва какво правят (на приятелите можем да доверим важна информация, която попринцип не искаме да е достъпна за останалия свят, но за тях правим изключение).

За какво ще ни бъдат полезни?

- В някои специфични случаи
- При предефиниране на оператори (дойдохме си на думата)

Приятелски функции

Декларация:

- Декларират се като нормална функция, само че в началото поставяме ключовата дума `firend`.
- Декларират се в тялото на класа, чиито `private` и `protected` данни искаме да достъпим.
- Могат да се декларират навсякъде в тялото на класа, без значение `private`, `public` или `protected` секция.
- Въпреки, че деклатацията на приятелските функции е в тялото на класа, те се водят **non-member functions**. Доказателство за това е, че не могат да използват `this` указател, към текущият обект (може да го пробвате в IDE-то си).

```
//Point.h

class Point{
private:
    double x;
    double y;
public:
    Point(double x, double y);
    void print() const;

friend double distance(const Point& a, const Point& b);
};
```

Дефиниция:

- Дефинира се като обикновена функция, без да се използва `friend`, както и не се обозначава scope (обхвата) (няма `Point::` преди името на приятелската функция distance), просто защото, както написах няколко пъти, тя не е член-функция на класа, тя му е "приятел".
- Дефинира се където си искате, може в тялото на класа, в .cpp файла на съответния клас, само трябва да се внимава с повторни дефиниции при includ-ите. В долния пример я дефинирам в .cpp файла.

```
//Point.cpp

Point::Point(double x, double y) : x(x), y(y) {};

void Point::print() const{
    cout << "(" << x << ";" << y << ")";
}

double distance(const Point& a, const Point& b){
    double distance = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
    return distance;
}
```

- Специфичното е, че приятелската функция може да има достъп до всички член-данни, независимо от идентификатора им (това за което ставаше дума през целия материал), не директно, а чрез обекти от класа, чийто приятел е. Какво ще рече това? В горния пример са подадени два обекта като аргументи на функцията и тя може да достъпи техните член-данни директно. Ако нямаме подаден като аргумент обект от класа, не може да ползвате `x` или `y` (спомнете си за `this` как работеше и, че приятелските функции нямат достъп до него).

Извикване:

- Както беше споменато, приятелската функция не е част от обекти на класа, тя му е само приятелче. Това означава, че и нейното извикване става самостоятелно, не е нужен обект.

```
#include "Point.h"

int main(){
    Point a(3,3);
    Point b(4,4);
    cout << distance(a,b);//правилно

    cout << a.distance(a,b);//грешно
    cout << b.distance(a,b);//грешно
}
```

Приложение:

След като уточнихме какво представляват приятелките функции, нека да видим и с какво могат да ни бъдат полезни с този пример:

Имаме задача да създадем функция, която намира разстоянието между две точки (ще си ползваме класа Point). Имаме няколко варианта как да имплементираме такава функция, без да нарушаваме принципа за енкапсулация:

1. Да създадем метод на класа Point, който приема като аргумент обект от класа Point и така ще правим разстоянието между;

```
class Point{
private:
    double x;
    double y;
public:
    //...
    double distance(const Point& rhs){
        return sqrt(pow(this->x - rhs.x,2) + pow(this->y - rhs.y,2));
    }
};
```

Става, но може и по-добър и красив код да се напише.

2. Да създадем функция, външна за класа Point, която прием като аргументи два обекта от класа Point и чрез *getter* да достъпим стойностите на двата обекта;

```
double distance(const Point& a, const Point& b){
    return sqrt(pow(a.get_x() - b.get_x(), 2) + pow(a.get_y() - b.get_y(), 2));
}

int main(){
    Point a(3,3);
    Point b(4,4);
    cout << distance(a,b);
    return 0;
}
```

Става, но имаме извикване на getter-и и това допълнително утежнява програмата

3. Да създадем приятелска функция на класа Point, която приема като аргументи два обекта от класа Point и може директно да работи с член-данните им

```
class Point{
private:
    double x;
    double y;
public:
    //...
friend double distance(const Point& a, const Point& b){
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
};

//в main

int main(){
    Point a(3,3);
    Point b(4,4);
    cout << distance(a,b);
    return 0;
}
```

Супер е вече

Заб: Направих ги inline, за да е по-лесно за четене, иначе вие знаета как е най-добре.

Ето и код в пълната му форма, същата задача, само че с четири точки от входа ги чете, направено с приятелската функция, вие може да си пробвате как работи и с другите варианти. Препоръчвам входа: (0,0), (4,0), (4,3), (0,3)

```
//////////////////////////////////////main.cpp//////////////////////////////
//////////

#include <iostream>
#include <cmath>
#include "Point.h"


int main(){
    Point arr[4];
    for(int i = 0; i < 4; i++){
        arr[i].read();
    }

    cout << "\nPoints:\n";
    for(int i = 0; i < 4; i++){
        arr[i].print();
        cout << "\n";
    }

    cout << "\nDistances:\n";
    for(int i = 0; i < 3; i++){
```

```
        for(int j = i+1; j < 4; j++){
            arr[i].print();
            cout << " | ";
            arr[j].print();
            cout << " | ";
            cout << distance(arr[i], arr[j]) << "\n";
        }
    }


    return 0;
}
```

```
//////////////////////////////////////Point.h//////////////////////////////
//////////

#ifndef POINT_H
#define POINT_H

#include <iostream>
#include <cmath>
using namespace std;

class Point{
private:
    double x;
    double y;
public:
    Point();
    Point(double x, double y);
    double get_x() const;
    double get_y() const;
    void set(double x, double y);
    void read();
    void print() const;

friend double distance(const Point& a, const Point& b);
};

#endif
```

```
//////////////////////////////////////Point.cpp//////////////////////////////
//////////

#include "Point.h"

Point::Point() : x(0), y(0){};

Point::Point(double x, double y) : x(x), y(y) {};

void Point::set(double x, double y){
    this->x = x;
    this->y = y;
}

double Point::get_x() const{
```

```
    return x;
}

double Point::get_y() const{
    return y;
}

void Point::read(){
    cout << "x:";
    cin >> x;
    cout << "y:";
    cin >> y;
}

void Point::print() const{
    cout << "(" << x << ";" << y << ")";
}

double distance(const Point& a, const Point& b){
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```