Обектно ориентирано програмиране

ОТ ЗАПИСИ КЪМ КЛАСОВЕ

Класове

Класовете са нови типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип данни.

Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп.

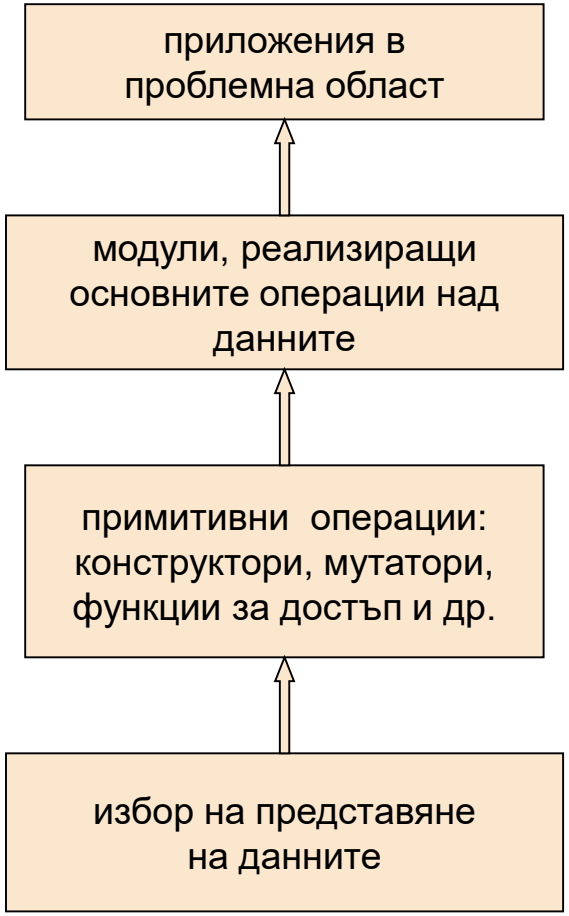
Абстракция със структури от данни

- Основен принцип на процедурното програмиране е модулния. Програмата се разделя на “подходящи” взаимосвързани части (функции, модули), всяка от които се реализира чрез определени средства.
- Подход *абстракция със структури от данни* - методите за използване на данните се разделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с “абстрактни данни” – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени **конструктори, мутатори** и **функции за достъп**, които реализират “абстрактните данни” по конкретен начин.

Абстракция със структури от данни

- Създаване на нови типове данни чрез групиране на съществуващи типове данни в **запис** (struct)
- Полетата на записа ще наричаме **член-данни**
- Създаваме операции, които работят върху член-данните
  - операциите „знаят“ как е представен обекта
- Външните функции, които използват новия тип работят с него чрез операциите
  - външните функции „не знаят“ как е представен обекта

Нива на абстракция



Нива на абстракция

- Добра реализация на подхода е тази, при която всяко ниво използва единствено средствата на предходното. Предимствата са, че възникнали промени на едно ниво ще се отразят само на следващото над него.
- Например, промяна на представянето на данните ще доведе до промени единствено на реализацията на някои от конструкторите, мутаторите или функциите за достъп.

Пример: рационални числа

Искаме да дефинираме тип данни “рационално число”, след което да го използваме за събиране, изваждане, умножение и деление на рационални числа.

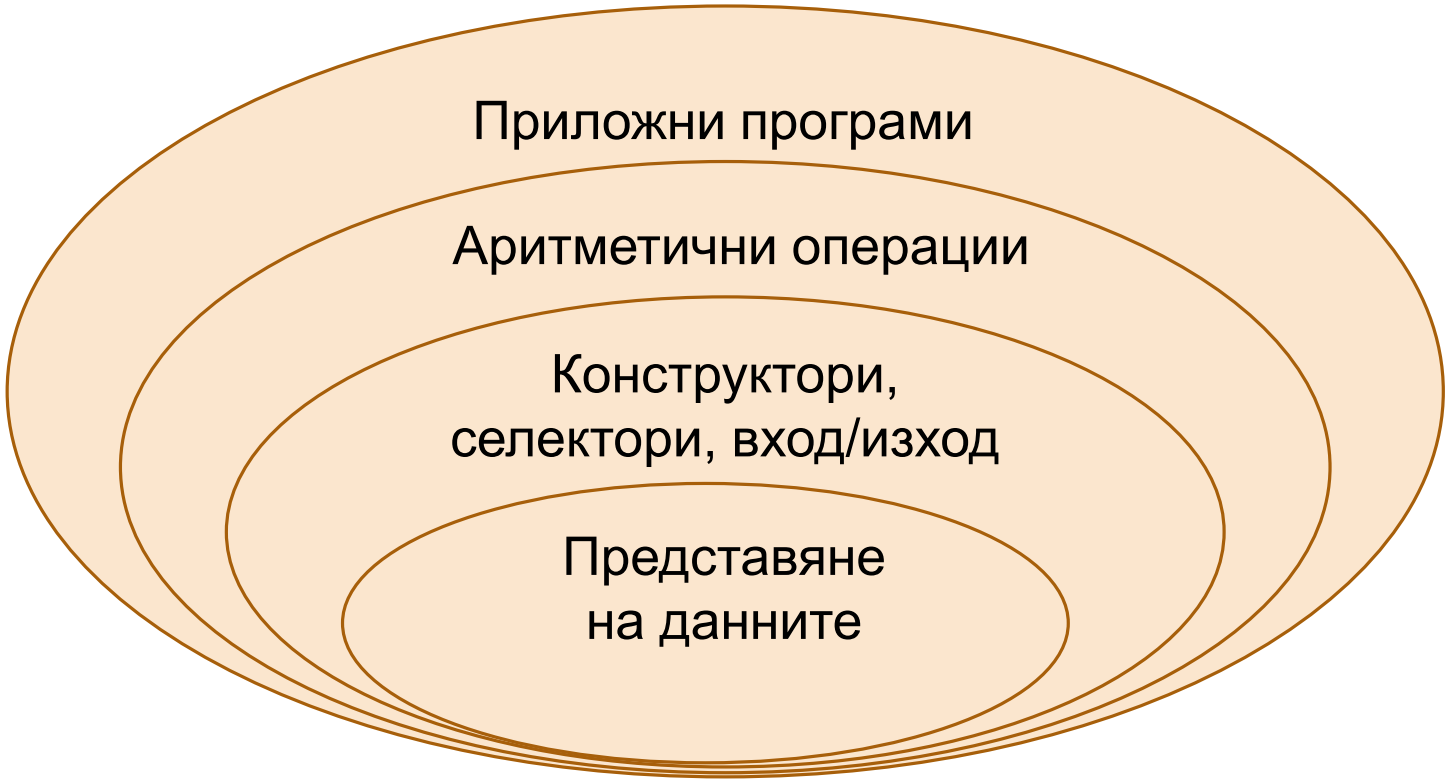
Примитивни операции за работа с рационални числа:

- конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател;
- извличане на числителя на дадено рационално число;
- извличане на знаменателя на дадено рационално число.

Пример: рационални числа

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател
- **Примитивни операции:** създаване на рационални числа, намиране на числител и на знаменател
- **Аритметични операции:** събиране, изваждане, умножение, деление
- **Други операции:** въвеждане и извеждане

Нива на абстракция



Пример: рационални числа

Избор на представяне на рационално число

Тъй като рационалното число е частно на две цели числа, можем да го определим чрез структурата:

```
struct Rational {
    int numer;
    int denom;
};
```

Пример: рационални числа

- Полето **numer** означава числителя, а полето **denom** – знаменателя на рационално число.
- Тези две полета се наричат **член-данни**, само **данни** или още **абстрактни данни** на структурата. Те определят множеството от стойности на типа **Rational**, който дефинираме.

Пример: рационални числа

Реализиране на примитивните операции

Като компоненти на структурата **Rational** ще добавим набор от примитивни операции: конструктори, мутатори и функции за достъп. Ще ги реализираме като член-функции.

а) конструктори

Конструкторите са член-функции, чрез които се инициализират променливите на структурата. Имената им съвпадат с името на структурата. Ще дефинираме два конструктора на структурата Rational:

- `Rational()` – конструктор без параметри и
- `Rational(int, int)` – конструктор с два цели параметъра.

Пример: рационални числа

Първият конструктор се нарича още **конструктор по подразбиране**. Използва се за инициализиране на променлива от тип Rational, когато при дефиницията ѝ не са зададени параметри. Ще го дефинираме така:

```
// конструктор по подразбиране
Rational::Rational() {
    numer = 0;
    denom = 1;
}
```

Пример: рационални числа

Пример: След дефиницията

```
Rational p = Rational();
```

или съкратено

```
Rational p;
```

p се инициализира с рационалното число 0/1.

Пример: рационални числа

Вторият конструктор

```
Rational::Rational(int n, int d) {
    numer = n;
    denom = d;
}
```

позволява променлива величина от тип `Rational` да се инициализира с указана от потребителя стойност.

Пример: рационални числа

Примери: След дефиницията

```
Rational p = Rational(1, 3);
```

p се инициализира с 1/3, а дефиницията

```
Rational q(2, 5);
```

инициализира q с 2/5.

Ще отбележим, че и двата конструктора имат едно и също име, но се различават по броя на параметрите си. В този случай се казва, че функцията Rational е **предефинирана (overloaded)**.

Декларацията на структура може да съдържа, но може и да не съдържа конструктори.

Пример: рационални числа

б) мутатори

Това са функции, които променят данните на структурата. Ще дефинираме мутатора read(), който въвежда от клавиатурата две цели числа и ги свързва с абстрактните данни numer и denom.

```
// мутатор (функция за промяна) чрез въвеждане
void Rational::read() {
    // 1/3
    cin >> numer;
    cin.ignore();
    cin >> denom;
}
```

Пример: рационални числа

След обръщението

```
p.read();
```

стойността на p *се променя* като полетата ѝ numer и denom се свързват с въведените от потребителя стойности за числител и знаменател съответно.

в) функции за достъп (селектори)

Тези функции **не променят** член-данните на структурата, а само извличат информация за тях. Последното е указано чрез използването на запазената дума **const**, записана след затварящата скоба на формалните параметри и пред знака ;.

Пример: рационални числа

Ще дефинираме следните функции за достъп:

```
int getNumerator() const;
int getDenominator() const;
void print() const;
```

Първата от тях извлича числителя, втората – знаменателя, а третата извежда върху екрана рационалното число numer/denom.

Пример: рационални числа

```
// селектор за числител
int Rational::getNumerator() const {
    return numer;
}
// селектор за знаменател
int Rational::getDenominator() const {
    return denom;
}
// функция за извеждане
void Rational::print() const {
    cout << getNumerator() << '/' << getDenominator();
}
```

Заглавни (header) файлове

- В заглавните файлове се пишат
  - дефинициите на записи и класове
  - декларации на функции
  - декларации на методи
- В заглавните файлове обикновено не се пишат
  - дефиниции на променливи
  - дефиниции на функции
  - дефиниции на методи
- Защо?
  - Абстракция: разделя се интерфейса от реализацията

Пример: рационални числа

```
// rational.h
struct Rational {
    int numer, denom;
    // конструктори
    Rational();
    Rational(int, int);
    // функции за достъп
    int getNumerator() const;
    int getDenominator() const;
    void print() const;
    // мутатор
    void read();
};
```

Пример: рационални числа

След направените дефиниции са възможни следните действия над рационални числа:

```
// p се инициализира с 0/1, q – с 1/6, а r – с 5/9
Rational p, q(1, 6), r = Rational(5, 9);
// p се извежда чрез полета на структурата Rational
cout << p.numer << "/" << p.denom << endl;
// q се извежда като се използват
// функциите за достъп до компонентите му
cout << q.getNumerator() << "/" << q.getDenominator()
     << endl;
// q се извежда чрез функцията за достъп print()
q.print();
// p се модифицира чрез мутатора read()
p.read();
```

Пример: рационални числа

Реализиране на правилата за рационално-цифрова аритметика

Като използваме дефинираните конструктори, мутатори и функции за достъп, ще реализираме функциите:

```
Rational add(Rational, Rational);
Rational subtract(Rational, Rational);
Rational multiply(Rational, Rational);
Rational divide(Rational, Rational);
```

извършващи рационално-числовата аритметика.

Пример: рационални числа

Функцията add може да се дефинира по следния начин:

```
Rational add(Rational p, Rational q) {
return Rational(p.getNumerator() * q.getDenominator()
        + p.getDenominator() * q.getNumerator(),
        p.getDenominator() * q.getDenominator());
}
```

Другите функции се реализират по аналогичен начин.

Пример: рационални числа

По подразбиране, членовете на структурата (член-данни и член-функции) са видими навсякъде в областта на структурата. Това позволява член-данните да бъдат използвани както от примитивните конструктори, мутатори и функции за достъп така и от функциите, реализиращи рационално-числова аритметика.

Например, функцията add, дефинирана по-горе може да се реализира и така:

```
Rational add(Rational p, Rational q) {
return Rational(p.numer * q.denom
        + p.denom * q.numer,
        p.denom * q.denom);
}
```

Пример: рационални числа

Нещо повече, освен чрез мутаторите, член-данните могат да бъдат модифицирани и от външни функции.

Последното **противоречи** на идеите на подхода абстракция със структури от данни, в основата на който лежи независимостта на използването от представянето на структурата от данни. Това води до идеята да се забрани на модулите от трето и четвърто ниво пряко да използват средствата от първо ниво на абстракция.

Пример: рационални числа

Езикът C++ позволява да се ограничи свободата на достъп до членовете на структурата като се поставят подходящи спецификатори на достъп в декларацията ѝ. Такива спецификатори са **private** и **public**. Записват се като етикети.

Всички членове, следващи спецификатора на достъп **private**, са достъпни само за член-функциите в декларацията на структурата.

Всички членове, следващи спецификатора на достъп **public**, са достъпни за всяка функция, която е в областта на структурата.

Ако са пропуснати спецификаторите на достъп, всички членове са **public** (както е в случая).

Има още един спецификатор на достъп – **protected**, който е еднакъв със спецификатора private, освен ако структурата не е част от йерархия на класовете, което ще разгледаме по-късно.

Пример: рационални числа

С цел реализиране на идеите на подхода абстракция със структури от данни, ще променим дефинираната по-горе структура по следния начин:

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

Пример: рационални числа

```
    // функции за достъп
    int getNumerator() const;
    int getDenominator() const;
    void print() const;
    // мутатор
    void read();
};
```

По такъв начин позволяваме член-данните **numer** и **denom** да се използват единствено от член-функциите на структурата **Rational**. Операторът

```
cout << p.numer << "/"  << p.denom << endl;
```

вече е недопустим.

Пример: рационални числа

Като се използват функциите за рационално-числова аритметика, могат да се реализират различни приложения.

Забелязваме обаче, че тази реализация не съкращава рационални числа.

За преодоляването на този недостатък е достатъчно да променим конструктора с параметри. За целта реализираме разделяне на числителя x и знаменателя y на най-големия общ делител на abs(x) и abs(y).

Новият конструктор има вида:

```
Rational::Rational(int n, int d) {
   if (n == 0 || d == 0) {
      numer = 0;
      denom = 1;
   }
```

Пример: рационални числа

```
    else {
        int g = gcd(abs(n), abs(d));
        if (n > 0 && d > 0 || n < 0 && d < 0) {
            numer = abs(n) / g;
            denom = abs(d) / g;
        }
        else {
            numer = -abs(n) / g;
            denom = abs(d) / g;
        }
    }
}
```

Пример: рационални числа

Ще отбележим, че ако в горната програма заменим запазената дума **struct** с **class**, програмата няма да промени поведението си. Така дефинирахме класа **rat**:

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

Пример: рационални числа

```
    // функции за достъп
    int getNumerator() const;
    int getDenominator() const;
    void print() const;
    // мутатор
    void read();
};
```

а дефиницията

```
Rational p, q = Rational(1, 7), r(-2, 9);
```

определя три негови **обекта**: p, инициализиран с рационалното число 0/1; q, инициализиран с 1/7 и r, инициализиран с -2/9.

Пример: рационални числа

- **struct** се заменя с **class**
- какво се променя?
  - почти нищо!
  - нивото на достъп по подразбиране в записите е **public**
  - а в класовете е **private**
- Защо са необходими и двете?

Пример: рационални числа

Спецификаторът **private**, забранява използването на член-данните numer и denom извън класа. Получава се *скриване на информация*, което се нарича още **капсулиране на информация**.

Член-функциите на класа Rational са обявени като **public**. Те са видими извън класа и могат да се използват от външни функции. Затова public-частта се нарича още **интерфейсна част на класа** или само **интерфейс**. Чрез нея класът комуникира с външната среда. Освен функции, интерфейсът може да съдържа и член-данни, но засега ще се стараем това да не се случва.