

Обектно ориентирано програмиране

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.

Пример за програма, която дефинира и използва клас

- 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.

Пример за програма ...

Нива на абстракция



Пример за програма ...

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 за програма ...

- 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, представлящи съответно неговите числител и знаменател;
 - извличане на числителя на дадено рационално число;
 - извличане на знаменателя на дадено рационално число.

Пример за програма ...

- Към тях ще добавим и функциите:
 - промяна на стойността на рационално число чрез въвеждане, например;
 - извеждане на рационално число.

Пример за програма ...

- четири нива на абстракция



Пример за програма ...

- *Избор на представяне на рационално число*
- Тъй като рационалното число е частно на две цели числа, можем да го определим чрез структурата:

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

Пример за програма ...

- полето `num` 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 `rat`, който дефинираме.

Пример за програма ...

- *Реализиране на примитивните операции*
- Като компоненти на структурата `rat` 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 `rat`:

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

Пример за програма ...

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

```
rat::rat()  
{  
    numer = 0;  
    denom = 1;  
}
```

Пример за програма ...

- **Пример:** След дефиницията

```
rat p = rat();
```

или съкратено

```
rat p;
```

p се инициализира с рационалното число 0/1.

- Вторият конструктор

```
rat::rat(int x, int y)
{
    numer = x;
    denom = y;
}
```

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

Пример за програма ...

- **Примери:** След дефиницията

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

p се инициализира с $1/3$, а дефиницията

```
rat 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 `rat` е **предефинирана**.
- Декларацията на структура може да съдържа, но може и да не съдържа конструктори.

Пример за програма ...

б) мутатори

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

```
void rat::read()
{
    cout << "numer: ";
    cin >> numer;
    do
    {
        cout << "denom: ";
        cin >> denom;
    } while (denom == 0);
}
```

Пример за програма ...

След обръщението

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

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

в) *функции за достъп*

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 за програма ...

- Ще дефинираме следните функции за достъп:

- ☐ `int get_numer() const;`

- ☐ `int get_denom() const;`

- ☐ `void print() const;`

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

Пример за програма ...

```
int rat::get_numer() const
{
    return numer;
}
int rat::get_denom() const
{
    return denom;
}
void rat::print() const
{
    cout << numer << "/" << denom << endl;
}
```

Пример за програма ...

```
struct rat
{
    int numer;
    int denom;
    // конструктори
    rat();
    rat(int, int);
    // мутатор
    void read();
    // функции за достъп
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

Пример за програма ...

- 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
rat p, q(1,6), r=rat(5,9);
// p се извежда чрез данновите полета на структурата rat
cout << p.numer << "/" << p.denom << endl;
// q се извежда като се използват
// функциите за достъп до компонентите му
cout << q.get_numer() << "/" << q.get_denom()
    << endl;
// q се извежда чрез функцията за достъп print()
q.print();
// p се модифицира чрез мутатора read()
p.read();
```

Пример за програма ...

- *Реализиране на правилата за рационално-цифрова аритметика*

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

```
rat sum(rat const &, rat const &);  
rat sub(rat const &, rat const &);  
rat prod(rat const &, rat const &);  
rat quot(rat const &, rat const &);
```

извършващи рационално-числовата аритметика.

Пример за програма ...

- Функцията `sum` може да се дефинира по следния начин:

```
rat sum(rat const& r1, rat const& r2)
{
    rat r(r1.get_numer()*r2.get_denom() +
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
}
```

- Другите функции се реализират по аналогичен начин.

Пример за програма ...

- 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 `sum`, дефинирана по-горе може да се реализира и така:

```
rat sum(rat const& r1, rat const& r2)
{
    rat r(r1.numer*r2.denom + r2.numer*r1.denom,
          r1.denom*r2.denom);
    return r;
}
```

Пример за програма ...

- 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 за програма ...

- 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 за програма ...

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

```
struct rat
{
    private:
        int numer;
        int denom;
```

Пример за програма ...

```
public:
    rat();
    rat(int, int);
    void read();
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

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

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

вече е недопустим.

- 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 $\text{abs}(x)$ и $\text{abs}(y)$.
- Новият конструктор има вида:

```
rat::rat(int x, int y)
{
    if (x == 0 || y==0)
    {
        numer = 0;
        denom = 1;
    }
```

Пример за програма ...

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

Пример за програма ...

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

```
class rat
{
    private:
        int numer;
        int denom;
    public:
```

Пример за програма ...

```
    rat();  
    rat(int, int);  
    void read();  
    int get_numer() const;  
    int get_denom() const;  
    void print() const;  
};
```

а дефиницията

```
    rat p, q=rat(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 за програма ...

- Спецификаторът `private`, забранява използването на член-данните `num` и `den` извън класа. Получава се *скриване* на информация, което се нарича още **капсулиране на информация**.
- Член-функциите на класа `rat` 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.

Пример за програма ...

- *Ще отбележим, че конструкторите се използват само когато се създават обекти. Опитите за **промяна** на обект чрез обръщение към конструктор предизвикват грешки.*

- **Пример:**

```
rat q(1, 7);      // коректно
q.rat();          // предизвиква грешка
q(2, 9);          // предизвиква грешка
q.rat(3, 4);      // предизвиква грешка.
```

- Забележете, езикът С++ дефинира структурите и класовете почти идентично. Съществена разлика е свързана със спецификаторите на достъп. По подразбиране членовете на структура имат **public** (публичен) достъп, а членовете на клас – **private** (частен) достъп.

Дефиниране на класове

- Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове.
- Дефинирането на един клас се състои от две части:
 - декларация на класа и
 - дефиниция на неговите член-функции (методи).

Декларация на клас

- Декларацията на клас се състои от *заглавие* и *тяло*.
- *Заглавието* започва със запазената дума **class**, следвано от името на класа.
- *Тялото* е заградено във фигурни скоби. След скобите стои знакът “;” или списък от обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп.

Декларация на клас ...

```
<декларация_на_клас> ::= <заглавие> <тяло>
<заглавие> ::= class [<име_на_клас>]
<тяло> ::= { <декларация_на_член>;
             {<декларация_на_член>;}
           };
<декларация_на_член> ::= [<спецификатор_на_достъп>:]
    <декларация_на_функция> |
    <декларация_на_данна>
<спецификатор_на_достъп> ::= private | public | protected
<декларация_на_функция> ::=
    [<тип>] <име_на_функция>(<параметри>)
<декларация_на_данна> ::= <тип> <име_на_данна>
    {, <име_на_данна>}
<тип> ::= <име_на_тип> | <дефиниция_на_тип>
```

Декларация на клас ...

- За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато.
- Имената на членовете на класа са **локални** за него, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена.
- Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

Декларация на клас ...

```
class point
{
private:
    double x, y;
public:
    point(double, double);
    void read();
    int get_x() const;
    int get_y() const;
    void print() const
};
```

Декларация на клас ...

- Препоръчва се член-данните да се декларират в нарастващ ред по броя на байтовете, необходим за представянето им в паметта. Така за повечето реализации се получава оптимално изравняване до дума.
- **Забележка:** Типът на член-данна на клас **не може** да съвпада с името на класа, но типът на член-функция на клас **може** да съвпада с името на класа.
- В тялото, някои декларации на членове могат да бъдат предшествани от **спецификаторите на достъп** `private`, `public` или `protected`. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се спецификатор за достъп е `private`. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.

Декларация на клас ...

- Препоръчва се, ако секция `public` съществува, да бъде първа в декларацията, а секцията `private` да бъде последна в тялото на класа.
- *Достъпът до членовете на класовете може да се разгледа на следните две нива:*
 - По отношение на *член-функциите* в класа е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим на пряк достъп**.

Декларация на клас ...

Поради тази причина функциите `rat()`, `read()`, `print()`, `get_numer()` и `get_denom()` са без параметри. Освен това член-функцията `print()` може да бъде дефинирана и по следния начин:

```
void rat::print() const
{
    cout << get_numer() << "/" <<
        << get_denom() << endl;
}
```

- По отношение на *функциите, които са външни за класа*, режимът на достъп са определя от начина на деклариране на членовете.

Декларация на клас ...

- Членовете на даден клас, декларирани като **private** са видими (достъпни) само в рамките на класа. Външните функции нямат достъп до тях.
Чрез използването на членове, обявени като **private**, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още **капсулиране на информацията**.
- Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като **public**.
- Освен като **private** и **public**, членовете на класовете могат да бъдат декларирани и като **protected**.

Дефиниране на методите на клас

- Дефинициите са аналогични на дефинициите на функции, но името на метода се предшества от името на класа, на който принадлежи метода, следвано от оператора за принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат **пълни**. (Операторът :: е ляво-асоциативен и с един и същ приоритет със (), [] и ->).

- **Дефиниция на метод на клас**

<дефиниция_на_метод_на_клас> ::=

[<тип>] <име_на_клас>::<име_на_функция>(<параметри>)

[**const**]

{ <тяло> }

<тяло> ::= <редица_от_оператори_и_дефиниции>

Дефиниране на методите на клас

- Ще отбележим, че дефиницията на конструктор **не започва** с <тип>
- Запазената дума `const` може да присъства само в дефинициите на функциите за достъп. Добрият стил на програмиране изисква използването на `const` в дефинициите на функциите за достъп и също в техните декларации. Ако се пренебрегне това изискване, могат да се създадат класове, които да не могат да се използват от други програмисти.

Дефиниране на методите на клас

- **Пример:** Нека искаме да използваме класа `rat`, но програмистът му е забравил или нарочно не е декларирач член-функцията `print()` като `const` и `rat` има вида:

```
class rat
{
    private:
        ...
    public:
        ...
        void print();
};
```

Дефиниране на методите на клас

- Нека декларираме класа `prat`, използващ класа `rat`, коректно, т.е. функциите за достъп обявяваме като `const`.

```
class prat
{
    private:
        int a;
        rat p;          // използване на класа rat
        ...
    public:
        ...
        void print() const;
};
```

Дефиниране на методите на клас

```
void prat::print() const
{
    cout << a << endl;
    p.print(); // тази print() е член-функцията
               // на класа rat
};
```

- Компиляторът ще съобщи за грешка в обръщението `p.print()`, защото `p` е обект на класа `rat`, а член-функцията `rat::print()` не е декларирана като `const`. Компиляторът предполага, че `p.print()` може да модифицира `p`. Но `p` е член-данна на `prat`, а `prat::print()` е `const`, с което твърдо е обещава да не го модифицира.



Дефиниране на методите на клас

- Обикновено дефинициите на методите са разположени веднага след декларирането на класа, на който те са членове.
- Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

Дефиниране на методите на клас

```
class rat
{
    private:
        int numer;
        int denom;
    public:
        rat()
        {
            numer = 0;
            denom = 1;
        }
        rat(int a, int b)
        {
            if (a == 0 || b==0)
            {
                numer = 0;
                denom = 1;
            }
        }
    }
```

Дефиниране на методите на клас

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

Дефиниране на методите на клас

```
void read()
{
    cout << "numer: ";
    cin >> numer;
    do
    {
        cout << "denom: ";
        cin >> denom;
    } while (denom == 0);
}
int get_numer() const
{
    return numer;
}
```

Дефиниране на методите на клас

```
int get_denom() const
{
    return denom;
}
void print() const
{
    cout << numer << "/" << denom << endl;
}
};
```

- В този случай обаче член-функциите се третират като **вградени (inline) функции**.

Дефиниране на методите на клас

- С цел повишаване на бързодействието, езикът C++ поддържа т.нар. вградени (inline) функции. Кодът на тези функции не се съхранява на едно място, а се копира на всяко място в паметта, където има обръщение към тях. Използват се като останалите функции, но при декларирането и дефинирането им заглавието им се предшества от модификатора inline.
- Ще добавим, че дефиницията на inline функция трябва да се намира в същия файл, където се използва, т.е. не е възможна разделна компилация, тъй като компилаторът няма да разполага с кода за вграждане.
- Използването на inline функции води до икономия на време, за сметка на паметта. Затова се препоръчва използването им само при “кратки” функции.

Дефиниране на методите на клас

- Често член-функциите се реализират като inline функции. Това увеличава ефективността на програмата, използваща класа.
- Декларацията на inline член-функции може да се осъществи и по следния начин:

```
class rat
{
private:
    int numer;
    int denom;
public:
    rat();
    rat(int, int);
```

Дефиниране на методите на клас

```
void read();
int get_numer() const;
int get_denom() const;
void print() const;
};
inline rat::rat()
{
    numer = 0;
    denom = 1;
}
inline rat::rat(int x, int y)
...
inline void rat::read()
...
```

Дефиниране на методите на клас

```
inline int rat::get_numer() const
...
inline int rat::get_denom() const
...
inline void rat::print() const
...
```

- В тялото на дефиницията на член-функция явно не се указва обектът, върху който тя ще се приложи. Този обект участва неявно - чрез член-данните на класа. Заради това се нарича **неявен параметър**, а член-данните – **абстрактни данни**.
- Връзката между неявния параметър и обект ще бъде показана в по-късно. Параметри, които участват явно в дефиницията на член-функция се наричат **явни**.
- *Всяка член-функция има точно един неявен параметър и нула или повече явни.*

Дефиниране на методите на клас

- Обикновено декларацията на един клас се поставя в .h файл, а дефинициите на методите на класа – в съответен .cpp файл. Това позволява лесно да се създават библиотеки от класове.

```
// файл rat.h
class rat
{
    private:
        int numer;
        int denom;
    public:
        rat();
        ...
};
```

Дефиниране на методите на клас

```
// файл rat.cpp
#include "rat.h"
rat::rat()
{
    numer = 0;
    denom = 1;
}
rat::rat(int x, int y)
...
void rat::read()
...
```

Дефиниране на методите на клас

```
// файл point.h
class point
{
    private:
        int x;
        int x;
    public:
        point(int, int);
        void read();
        ...
};
```

Дефиниране на методите на клас

```
// файл point.cpp
#include "point.h"
point::point(int a, int b)
{
    x = a;
    y = b;
}
void point::read()
...
```

Дефиниране на методите на клас

```
// файл prog.cpp
#include <iostream.h>
#include "rat.h"
#include "point.h"
void main()
{
    rat q(1,3);
    point a(5,5);
    ...
}
```

Област на класовете

- За разлика от функциите, класовете могат да се декларират на различни нива в програмата: *глобално* (ниво функция) и *локално* (вътре във функция или в тялото на клас).
- Областта на глобално деклариран клас започва от декларацията и продължава до края на програмата.
- Ако клас е деклариран във функция, всички негови член-функции трябва да са `inline`. В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

Област на класовете

```
void f(int i, int* p)
{
    int k;
    class CL
    { public:
        // всички методи са дефинирани в тялото на класа
        ...
    private:
        ...
    };
    // тяло на функцията f
    CL x;
    ...
}
```

Област на класовете

- Областта на клас, дефиниран във функция, е функцията. Обектите на такъв клас са видими само в тялото на функцията.
- Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

```
void f(...)  
{  
    ...  
    class CL  
    {  
        // не може да се използва функцията f  
    };  
    ...  
}
```


Обекти

- След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат **обекти**. Връзката между клас и обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество **компоненти** (член-данни и член-функции).

Обекти

■ Дефиниция на обект на клас

<дефиниция-на_обект_на_клас> ::=

<име_на_клас> <обект>

[=**<име_на_клас>(<фактически_параметри>)**]

{, <обект>[=**<име_на_клас>(<фактически_параметри>)**] }

{, <обект>(**<фактически_параметри>**)}

{, <обект> **=** <вече_дефиниран_обект>;

<обект> ::= <идентификатор>

Обекти

- *Когато за даден клас явно са дефинирани конструктори, при всяко дефиниране на обект на класа те автоматично се извикват с цел да се инициализира обектът. Ако дефиницията е без явна инициализация (например `rat p;`), дефинираният обект се инициализира според дефиницията на конструктора по подразбиране, ако такъв е определен, и се съобщава за грешка в противен случай. Ако дефиницията е с явна инициализация, обръщението към конструкторите трябва да бъде коректно.*

- **Пример:**

```
rat p, q(2,3), r=rat(3,8);  
rat t=q;
```

Обекти

- *Когато за даден клас явно не е дефиниран конструктор, компилаторът автоматично генерира **подразбиращ се конструктор**. Този конструктор изпълнява редица действия, като заделяне на памет за обектите, инициализиране на някои системни променливи и др. Дефиницията на обект от този клас трябва да е без явна инициализация.*


- **Пример:**

```
#include <iostream.h>
class pom
{
private:
    int a;
```

Обекти

```
public:
    int b;
    void read();
    void print() const;
};

void main()
{
    pom x;    // инициализация според подразбиращия се
              //конструктор, генериран от компилатора на C++
    x.read();
    x.print();
}
```



```
void pom::print() const
{
    cout << "a= " << a << " b=" << b << endl;
}
void pom::read()
{
    cout << "a= ";
    cin >> a;
    cout << "b= ";
    cin >> b;
}
```

Обекти

- Декларацията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

```
rat p, q(2, 3), r = rat(3, 8);
```

заделя за обектите p, q и r по 8 байта ОП (по 4В за всяка от данните им numer и denom).

- Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната или метода, разделени с точка. Изключение от това правило правят конструкторите.

Обекти

■ Достъп до компонент на обект

<компонент_на_обект> ::=

<обект>.<данна> |

<обект>.<име_на_член_функция>() |

<обект>.<име_на_член_функция>(<параметри>)

<име_на_член_функция> е <идентификатор>, означаващ име на мутатор или име на функция за достъп.

Обекти

■ Пример:

```
rat p(1,2), q;
```

```
p.get_numer()    // достъп до член-функцията  
                  // get_numer() за обекта p
```

```
q.get_numer()    // достъп до член-функцията  
                  // get_numer() за обекта q
```

- Ще отбележим също, че на практика обектите `p` и `q` нямат свои копия на метода `get_numer()`. И двете обръщения се отнасят за един и същ метод, но при първото обръщение се работи с данните за обекта `p`, а при второто – с данните за обекта `q`.

Обекти

- При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта.
- Естествено възниква въпросът *по какъв начин методите на един клас “разбират” за кой обект на този клас са били извикани*. Отговорът на този въпрос дава указателят `this`. Всяка член-функция на клас поддържа допълнителен формален параметър - указател с име `this` и от тип `<име_на_клас>*`.

Обекти

- За да разберем точно как става това, ще разгледаме как компилаторът на C++ обработва член-функция и обръщение към член-функция на клас. Извършват се следните преобразувания:
 - а) Всяка член-функция на даден клас се транслира в обикновена функция с уникално име и един допълнителен параметър – указателят `this`.

Пример: Функцията

```
void rat::print()  
{  
    cout << numer << "/" << denom << endl;  
}
```

Обекти

се транслира в

```
void print_rat(rat* this)
{
    cout << this->numer << "/"
          << this->denom << endl;
}
```

б) Всяко обръщение към член-функция се транслира в съответствие с преобразуванието от а).

Пример: Обръщението

```
p.print();
```

се транслира в

```
print_rat(&p);
```

Обекти

- Указателят `this` може да се използва явно в кода на съответната член-функция.
- Като приложение на указателя `this` ще реализираме функцията

```
rat sum(rat const &, rat const &);
```

като член-функция на класа `rat`. За целта ще включим декларацията ѝ

```
rat sum(rat const &, rat const &);
```

в `public`-секцията на тялото на `rat` и ще я дефинираме по следния начин:

Объекти

```
rat rat::sum(rat const & r1, rat const & r2)
{
    numer = r1.numer*r2.denom+r2.numer*r1.denom;
    denom = r1.denom*r2.denom;
    return *this;
}
```

```
rat p=rat(1,4), r(1,2), q=rat(1,4);
...
r.sum(p.sum(p, r), q);
r.print();
p.print();
```

Обекти

- *Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация.*

- **Пример:** Допустими са дефинициите

```
rat p, q(4,5), r=q;
```

```
p = q;
```

```
...
```

```
r = p;
```

- При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
```

```
r.denom = p.denom;
```