Тема 16. Обектно-ориентирано програмиране. Основни принципи. Класове и обекти. Наследяване и капсулация

0. Абстракция със структури от данни. Класове и обекти.

Основната идея на ООП е представяне на частите от решаваната задача като набор от **обекти**, които включват в себе си **данни**(член-данни, атрибути, полета) и **методи**(член-функции) за обработката на тези данни. Еднотипните обекти се групират в **класове**.

Клас - множество от функции и променливи, които са обвързани ***в обща логическа структура***.
Обект - съвкупност от елементи и тяхното поведение.

Основните принципи на ООП

- **Абстракция** със структури от данни - **представянето на данните е отделено от използването им**
  - капсулация - ограничаване на достъпа
- **Отворена рекурсия** - методите работят със собствените данни на обекта
- **Наследяване** - един клас от обекти може да разширява друг вече съществуващ клас като използва наготово функционалността му
- **Генеричност** - обработване на различни класове обекти по ***универсален начин***
- **Полиморфизъм** - обработване на различни класове обекти по ***специфичен за тях*** начин
- **Динамично свързване** - извиканият метод се определя ***по време на изпълнение***, в зависимост от обекта, а не от класа, на който принадлежи

1. Декларация на клас и декларация на обект.

Ще разгледаме класовете в C++.

```
class <име> {
    <тяло>
} inst1, inst2, ..., instn;
```

Декларацията на класа се състои от име на класа и тяло, съдържащо декларации на член-функции и член-данни.
Дефиницията на класа се състои освен от декларацията му, и от дефиницията на неговите методи, която може да е **inline** в тялото на класа, или извън декларацията на класа, когато имената им се предхождат от `<име>::` .

Член-данните и член-функциите могат да бъдат дефинирани с различни спецификатори за достъп - `private` , `public` , `protected` .

- `private` - позволен е само вътрешен достъп на методите и полетата, обявени за `private` .
- `public` - позволен е и вътрешен и външшен достъп
- `protected` - вътршен достъп в класа и наследниците му.

След като клас е дефиниран, може да се създават негови инстанции(екземпляри, обекти от класа).

Създаването на обекти е свързано с **инициализиране** на паметта на обекта по такъв начин, че той да бъде във валидно състояние, т.е. ***да бъдат в сила ограниченията, наложени от инварианта на класа***.

Основни видове конструктори

```
<конструктор> ::=
<име-на-клас>::<име-на-клас>(<параметри>)
[ : <член-данна>(<израз>) {, член-данна(<израз>)}] {<тяло>}
```

1. Обикновен конструктор с параметри
2. Конструктор по подразбиране - конструктор без параметри; важно напр. при заделяне на масиви с new от елементи от класа
3. Конструктор с параметри по подразбиране - като 1, но има зададени параметри по подразбирane
4. Конструктор за копиране - за инициализация на обект се използва като образец друг обект, приема параметър от тип `<име на клас> const&`
5. Системно генерирани конструктори

- по подразбиране - ако в един клас не се дефинира нито един конструктор, системно се създава конструктор по подразбиране с празно тяло
- за копиране - ако не е дефиниран конструктор за копиране, то системно се създава такъв, който копира дословно полетата на образеца(полетата, които са обекти на класове се копират посредством копиращите си конструктори, а другите чрез копиране на стойностите в паметта)

6. Конструктор за преобразуване на тип(конвертиращ конструктор) - конструкторите с точно един параметър са специални, т.к. задават **правило** за конструиране на обект от класа по обект от друг клас, или от стойност от вграден тип. Навсякъде, където се очаква обект от клас A, но се подава стойност от тип B, C++ се оптива да използва конструктор за преобразуване на тип от вида A(B)

```
class Point2D {

private:
    double x = 0; // системният конструктор по подразбиране ще инициализира с тези стой
    double y = 0;

public:
    Point2D(int x, int y): x(x), y(y) {}
    void print() const {
        std::cout << "Point2D(" << x << ", " << y << ")\n";
    }
}



int main() {
    Point2D p(1, 2);
    p.print();
}
```

Управление на динамичната памет и ресурсите(RAII)

Жизнен цикъл на обект

- за обекта се заделя памет и се свързва с неговото име, извиква се подходящ конструктор на обекта
- работа с обекта - достъп до компоненти на обект, изпълняване на операции
- достига се края на областта на действие на обекта
- извиква се деструкторът на обекта

- заделената за обекта памет се освобождава

Дефиниране на деструктор

```
<име-на-клас::~<име-на-клас>() {<тяло>}>
```

Всеки клас може да има **точно един** деструктор, а ако не бъде дефиниран явно, се дефинира системен такъв с празно тяло. Съществено е, че ако обектът използва динамично заделен външен ресурс, то системният дескструктор няма да я освободи.

Това е идеята на **RAII**(Resourse Acquisition Is Initialization) стратегията за управлението на ресурсите и жизнения цикъл на обектите в в C++. ***Ако обект използва външна памет и ресурси, то техния жизнен цикъл е обвързан с жизнения цикъл на обекта.***

Когато се стигне до края на областта на действие на обекта:

- вика се неговия деструктор
- изпълнява се тялото му
- деструкторите на полетата, които са обекти, се извикват неявно в ред, обратен на реда им на конструиране
- тогава вече обектът се счита за унищожен

Методи - декларация, предаване на параметри, връщане на резултат.

```
class Test {
    <тип на връщане> <име> (<параметри>);
}
```

- Член-функциите имат достъп до всички полета на класа.
- Член-функциите се преобразуват до външни ф-ии с един доп. парам. `this` - указател към текущия обект(обекта, от който е извикана функцията).

Тъй като те са функции, това което важи за функциите в C++ относно предаване на параметри и връщане на резултат, е в сила и за методите.

Можем да разгледаме няколко вида методи:

- **конструктори** - функции за построяване на обекти

- **селектори** - функции за достъп до компоненти на обекти; обикновено връщат const reference или указател към съответния атрибут или копие на стойността
- **мутатори** - функции за промяна на състояние на обекти, но под контрол

Наследяване. Производни и вложени класове. Достъп до наследените компоненти. Капсулация и скриване на информацията. Статични полета и методи.

Наследяване

Има различни начини за получаване на данните и поведението на друг клас.
Един по-прост такъв е влагането(композиция) - напр. класът A съдържа обект от класа B като поле. Тогава отношението е от типа ***has-a***.

// Ако се има предвид nesting - класът B е вложен в класът A, когато е дефиниран в рамките на A.

Друг е наследяването, в C++ механизмът е class-based inheritance, който поражда ***is-a*** отношение между производни(детски, наследници) и родителски(базови, основи) класове. Класът-наследник получава всички член-данни и методи на родителя.

Дефинирането на компоненти, чието име съвпада с компонента на основен клас, наричаме **предефиниране**.

```
class Der: Base {
....
};
```

***Достъп до наследените компоненти*:**

Модификатори за достъп на членове на клас

- `private` - достъп само в класа
- `protected` - достъп само в класа и наследниците
- `public` - външен и вътрешен достъп

Видове наследяване
3 вида наследяване - public/private/protected

Ще ги илюстрираме с пример на C++:

```
class A {
public: // by default - private in classes, public in structs
    int x;
protected:
    int y;
private:
    int z;
};

class B: public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C: protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
}

class D: private A {
    // x is private
    // y is private
    // z is not accessible from D
}
```

Капсулация и скриване на информация

Принципът на капсулацията е, че се разделя описанието на типа данни от конкретната му реализация. Обектите **скриват** вътрешното си състояние - данните от които се състоят, връзките между тях и как те си взаимодействат. Само самите обекти могат да модифицират състоянието си директно, запазвайки инварианта на структурата данни. Осъществява се чрез групиране на данни и функции в клас и с ***модификаторите за достъп.***

Статични полета и методи

Статични полета и методи

- полета и методи, които не са обвързани с конкретен обект, но смислово са обвързани с класа(т.е. не зависят от **this** указател).

```
struct Test{
public:
    int a;
    static int b; // обща за всички обекти от класа,
    // жизнен цикъл като на глобална променлива
    void f() {}
    static void g() {}
};

int Test::b = 10;
Test::b++;
Test::g();
// можем и през обекти да ги достъпваме(съгласно спецификаторите за достъп),
//но те пак няма да зависят от this
```