16. ООП. Основни принципи. Класове и обекти. Наследяване и капсулация

1. **Абстракция със структури от данни. Класове и обекти**
   Абстракцията със структури от данни е основен принцип на ООП, според който представянето на данните е отделено от използването им. Предимства на абстракцията са изолирането на промените и

   ООП се фокусира на създаването на програмно-дефинирани типове данни, които съдържат и свойства и поведение.
   Класовете в ОО езици представляват шаблони, по които се създават инстанции на класа. Инстанциите наричаме **обекти**. Всеки клас дефинира член-функции (методи) и член-данни (полета). Полетата са променливи, които са асоциирани с конкретен обект на класа (освен когато са static). Методите определят какво е поведението на обектите на класа по време на живота им в процеса.

2. **Декларация на клас и декларация на обект. Основни видове конструктори**
   Декларацията на клас се състои от име на класа и тяло, съдържащо декларация на член-данни (полета) и член-функции (методи). Синтаксис:
   class ⟨име_на_класа⟩ {
   ⟨тяло⟩
   } $[a_1, a_2, \dots, a_n]$;

   Дефинирането на класа се състои освен от декларацията му, и от дефиницията на неговите методи, което може да е inline (в тялото на класа) или извън класа, когато имената им се предхождат от ⟨име_на_класа⟩ ::, където :: е бинарен оператор за достъп.

   Член-данните и член-функциите могат да бъдат декларирани с различни модификатори за достъп - private, protected и public. Private забранява външен достъп до данните/методите, декларирани в класа. Protected забранява външен достъп до данните/методите декларирани в него, с изключение на класовете-наследници. Public позволява всякакъв достъп до данните/методите.

   След като клас е дефиниран, може да се създават негови инстанции (екземпляри). Връзката между клас и обект в C++ е като тази между тип и променлива, но обектът се състои от множество компоненти.

   Създаването на обекти е свързано с инициализиране паметта на обекта, задаване на начални стойности и др. дейности, наречени инициализация и изпълнявани от специален вид член-функции на класовете - **конструкторите**. Конструкторът има следните особености:
   - Името му съвпада с името на класа
   - Типът на резултата е this (указател към създадения обект) и явно не се указва
   - Изпълнява се автоматично при създаване на обекти
   - Не може да се извика явно

   Дефиниция на конструктор:
   ⟨име_на_класа⟩ (⟨параметри⟩) [: ⟨член-данна⟩ (⟨израз⟩) {, ⟨член-данна⟩ (⟨израз⟩)}]
   {⟨тяло⟩}

   Във всеки клас може да се дефинира един или няколко конструктора. Видове конструктори:
   - *Обикновен конструктор с параметри*
   - *Конструктор с параметри по подразбиране* - подобно на функциите с параметри по подразбиране, може да зададем стойности по подразбиране на някои или всички параметри на функцията. Ако един параметър е с подразбиране, то и тези след него също трябва да са.
   - *Системно генериран конструктор по подразбиране* - Когато няма дефинирани конструктори се създава такъв, реализиращ действия като заделяне на памет за член данните, инициализиране на някои системни променливи и т.н.
   - *Конструктор за копиране* - за инициализация на обект се използва като образец друг обект. Приема параметър от тип ⟨име_на_клас⟩const &. Ако е дефиниран конструктор за копиране, то компилаторът го ползва. Иначе се създава системен такъв, който копира дословно полетата на образеца.

- *Конструктор за преобразуване на тип* - конструкторите с точно един параметър са специални, защото задават правило за конструиране на обект от класа по обект от друг клас или стойност от вграден тип. Навсякъде, където се очаква обект от клас A, но се подава стойност от тип B, C++ се опитва да използва конструктор за преобразуване на тип от вида A(B).
- *Преместващ конструктор* - приема параметър от тип ⟨име_на_клас⟩ &, като цели да премести съдържанието на подадения обект в новосъздаващия се. Обектът-параметър остава празен и вероятно няма да се ползва повече. Компилаторът системно генерира такъв конструктор по подразбиране

Пример:

```
class Pair
{
private:
        int x { };
        int y { };
public:
        Pair(int x, int y) : x{x}, y{y} { }

        void print( ) const {
                std :: out ≪ Pair( ≪ x ≪. ≪ y ≪ )\n);
        }
};
```

```
int main( )
{
        Pair pair{1,2};
        pair.print( );
}
```

3. Управление на динамична памет и ресурсите ("RAII")

Областта за динамична памет (heap) е набор от свободни блокове памет. Динамичната памет може да бъде заделена и освободена по всяко време на изпълнението на програмата. Програмата може да заяви блок с произволна големина. ОС се грижи за управлението на динамичната памет.

Програмистът носи отговорност за правилната работа с динамичната памет.
Заделена динамична памет остава непокътната до освобождаването ѝ с delete или до завършване на програмата. След приключване на програмата, цялата заделена от нея памет се освобождава от ОС.

RAII (Resource acquisition is initialization) описва поведение на програмен език, при което притежаването на ресурс е инвариант на класа и е свързано с продължителността на живота на обекта. Заделянето (или придобиването) на ресурса се извършва по време на създаването на обекта (инициализацията) от конструктора, а освобождаването се извършва по време на разрушаването на обекта от деструктора.
В C++ се реализира с т.нар. голяма четворка: конструктор, копиращ конструктор, деструктор и оператор=. Функциите на голямата четворка се генерират системно, ако не ги напишем. Пишем ги, когато трябва да управляваме външни за обекта ресурси (като динамична памет)

Деструкторът представлява функция, която се извиква автоматично при унищожаване на обекта след достигане на края на областта на действие на обекта или извикване на delete/delete[]. Той е противоположен на конструктора и служи за освобождаване на използваните ресурси. Синтаксис:
⟨име_на_класа⟩ :: ~⟨име_на_класа⟩( ) {⟨тяло⟩}
Всеки клас може да има точно един деструктор. Ако не бъде дефиниран явно, системно се генерира деструктор с празно тяло.

Оператор= представлява предефиниране на функционалността на оператора = за обекти от дадения клас.

При управлението на динамична памет е нужно да знаем коя част от програмата носи отговорност да освободи динамичната памет, когато вече не е нужна.
- Вариант 1 - собственост. Един от указателите се счита за собственик, а другите - за ползватели. Динамичната памет се използва през собственика
- Вариант 2 - умни указатели. Споделена собственост между няколко указателя, като се поддържа общия брой на всички указатели към дадена памет, когато броят стане 0, паметта се освобождава

4. **Методи - декларация, предаване на параметри, връщане на резултат. Предефиниране на операции**

Вече описахме, че член-функциите (методите) се дефинират или в тялото на класа или чрез оператора :: като за тях също важат модификаторите за достъп.

При извикване на метод, имаме достъп до полетата без да се указва обекта. Това се случва, защото при извикване на метод се създава автоматично специален константен указател с име this:

⟨име_на_класа⟩ * const this

Той сочи към обекта, за който е извикан метода. Компилаторът скрито от нас превежда методите т.ч. да получават this като първи параметър и всяко поле на обекта в тялото се достъпва през this.

Съществува специален тип методи - константни методи, при които методът гарантира, че няма да променя състоянието на обекта или да извиква не-константни методи. Синтаксис:

⟨тип⟩ ⟨име⟩⟨операция⟩(⟨параметри⟩) ***const*** {⟨тяло⟩}

Предаването на параметри и връщането на резултат е сходно като при обикновените функции. Параметрите могат да се подават по име или по стойност. Методите могат да връщат само един обект като резултат.

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят с обекти от произволен клас. Не могат да бъдат предефинирани следните операции: ? : , ::, . , sizeof, #, ##

Синтаксис на метод за предефиниране на операция:

⟨тип⟩ *operator*⟨операция⟩([⟨тип⟩]) [*const*] {⟨тяло⟩}

Пример с класа Pair:

$Pair\ operator * (Pair\ const\&\ p)\ const$ {
    $return\ Pair(x * p.x, y * p.y);$
}

5. **Наследяване. Производни и вложени класове. Достъп до наследените компоненти**

Вложен клас представлява добавянето на клас като поле в друг клас. То изразява връзка от типа "has-a"(има). Основна идея на наследяването е създаване на нови класове чрез използване на атрибути и поведение на съществуващи класове. То изразява връзка от типа "is-a"(Е).

Класът, който наследяваме се нарича основен, родителски или базов.
Класът, който наследява, наричаме производен, наследник или подклас.

Синтаксис на наследяването:

class ⟨име⟩ : [⟨видимост⟩] ⟨базов клас⟩ {, [⟨видимост⟩] ⟨базов клас⟩}
    {⟨тяло⟩};

Видимостта може да има 3 стойности:

private, *protected*, *public*.

Видимостта определя достъпът на функции, които не са методи на класа и не са приятелски

Ако достъпът е	public	protected	private
...а наследяването е	то външният достъп е		
public	неограничен	за наследници	забранен
protected	за наследници	за наследници	забранен
private	забранен	забранен	забранен

По подразбиране видимостта е private.

При наследяване, производният клас получава от основния клас всички негови полета, методи и достъп до неговите public и protected компоненти.
Производният клас не получава от основния достъп до приятелите му; private компоненти, но ги съдържа.
Производният клас може да дефинира свои методи и полета, дори и те да са със същите имена като в основния клас.

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

При наследяване, обект от производния клас съдържа указател към началото на базовия клас.

Когато се инстанцира обект от производен клас се случват следните неща:
- Заделя се памет за обект от производния клас (достатъчно и за базовата и за производната част)
- Извиква се подходящият конструктор на производния клас
- Първо се конструира обект от базовия клас, ако не е специфициран конструктор се използва този по подразбиране
- Инициализират се променливите от инициализиращия списък
- Изпълнява се тялото на конструктора

Пример за наследяване:

```
class Base {
private:
    int id { };

public:
    Base (int id = 0) : id {id} {
        std :: cout << "Base\n";
    }

    int getID( ) const { return id; }
};
```

```
class Derived : public Base {
private:
    double cost{ };

public:
    Derived(double cost = 0.0, int id = 0)
        : Base{id}, cost{cost} {
            std :: cout << "Derived\n";
        }
    double getCost( ) const { return cost; }
};
```

Сега при изпълнение на:

```
int main( ) {
    Derived derived{1.3, 5};
    std :: cout << "Id: " << derived. getId( ) << '\n';
    std :: cout << "Cost: " << derived. getCost( ) << '\n';

    return 0;
}
```

Ще изведе:

```
Base
Derived
Id: 5
Cost: 1.3
```

Производният клас (B) се счита за подтип на основния (A). Така:
- Всеки обект от тип B може да се разглежда като от тип A
- Някой обект от тип A може да се окаже от тип B.
- B е частен случай/разширение на A, B има повече информация от A
- Обекти, указатели и препратки от B се преобразуват в обекти, указатели и препратки от A неявно

Пример:

```
Derived d; Derived * pd = &d; Derived& rd = d;
Base p = d;
Base * pb = &h; pb = pd; pb = &rd;
```

Наследяване и влагане, *прилики*:
- Физическото представяне е почти еднакво
- Забранени са циклични зависимости
- Получават се полетата и методите на използвания клас

Наследяване и влагане, *разлики*:
- Наследените методи се викат автоматично
- Може да се влагат няколко обекта от един и същи клас
- Влагането може да бъде динамично (чрез указател)

6. Капсулация и скриване на информация

Принципът на капсулацията е, че се разделя (абстрахира) 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 директно, по този начин се грижат за запазване на инвариантата на структурата от данни, репрезентираща състоянието. Осъществява се чрез групиране на данни и функции в клас и с модификаторите за достъп private, protected и public.

7. Статични полета и методи

Стандартно, при създаване на инстанция, всеки обект получава свое копие на всички нормални полета.

Полета на класа могат да бъдат направени статични, използвайки ключовата дума *static*. Статичните полета са споделени между всички обекти на дадения клас. Те не се асоциират с конкретен обект и съществуват дори и да няма инстанцирани обекти от конкретния клас. На практика те са глобални променливи, които живеят в областта на класа. Можем да ги инициализираме използвайки оператора :: или при декларацията им, използвайки директно инициализиране с {}.

Пример:

```
class Foo {
public:
    static const int sc_value{4};
    static int s_value;
};

int Foo :: s_value {1};
```

Методите също могат да бъдат статични, тогава те няма да бъдат асоциирани с конкретен обект и могат да бъдат извиквани директно, използвайки името на класа и оператора за достъп ::. Синтаксис:

static ⟨тип⟩ ⟨име⟩(⟨параметри⟩) {⟨тяло⟩} - дефиниция на статична функция

⟨име_на_класа⟩ :: ⟨име_на_статичен_метод⟩(⟨аргументи⟩); - извикване на статична функция

Те могат да бъдат извиквани и от инстанции на класа.

Тъй като те не са асоциирани с конкретен обект, нямат и this указател. Те могат да достъпват други статични компоненти (полета и методи), но не и нормални такива.

Пример:

```
class IDGenerator {
private:
    static inline int s_id {1};
public:
    static int getNextID( );
};

int IDGenerator :: getNextID( ) { return s_id ++; }

int main( ) {
    for (int i = 0; i < 5; ++i) {
        std :: cout <<  IDGenerator :: getNextID( ) <<' \n';
    }
    return 0;
}
```

Ще изведе:

```
1
2
3
4
5
```