12. Обектно ориентирано програмиране на базата на CPP: Основни принципи. Класове и обекти. Конструктори и деструктори. Оператори. Производни класове и наследяване.

Анотация: Изложението по въпроса трябва да включва следните по-съществени елементи:

1. Основни принципи на обектно ориентираното програмиране;
2. Класове. Дефиниране на класове. Област на класове. Обекти;
3. Конструктори. Дефиниране на конструктори. Видове конструктори;
4. Указатели към обекти. Масиви и обекти. Динамични обекти;
5. Деструктори. Създаване и разрушаване на обекти на класове;
6. Оператори. Предефиниране на оператори;
7. Производни класове. Дефиниране. Наследяване и достъп до наследените компоненти.

Обектно ориентиранатото програмиране се основава на идеята за обект. Обектът е единица, която притежава данни и има поведение (извършва операции). Данните се наричат свойства на обекта, а операциите включват достъп и модифициране на тези свойства както и други функции, които изобразяват поведението на обекта.

1. Основни принципи на обектно ориентираното програмиране.

1.1. Енкапсулация. Обектите скриват вътрешното си състояние – данните от които се състои, връзките между тях и как те си взаимодействат. Само те могат да го модифицират директно, като по този начин се грижат за възстановяване на инвариантата на структурата от данни репрезентираща състоянието при евентуално нарушаване. Клиентският код има достъп до вътрешното състояние на обекта само през специфични методи на самия обект. В C++ това се осъществява чрез групирането на данни и функции в клас (class) или структура (struct) и с модификаторите за достъп private, protected и public.

1.2. Абстракция. Обектите показват на външния свят само интерфейс, но не и допълнителни детайли за начина му на функциониране. Интерфейсът, който се предоставя на потребителя трябва да е минимален по включване относно пълнотата на функциониране на обекта. Промяната в даден обект е локална и не се отразява върху обектите, които го използват. Това прави кода по-лесен за поддръжка. В C++ принципът на абстракция се осъществява чрез модификаторите за достъп, както и чрез специални header файлове.

1.3. Наследяване. Обект (клас или структура) може да наследява друг обект. Класовете които се наследяват се наричат базови, а класовете, които ги наследяват – дериватни. Дериватните класове ще наследят всички данни и функционалност на базовите класове, но достъпа им до тях ще зависи от спецификатора за достъп на дериватния клас. Дериватните класове не се ограничават до член данните и методите на базовия клас – те могат да дефинират свои данни, както и поведение. Най-голямите предимства на наследяването са преизползването на код (позволява на програмистите да преизползват вече съществуващи функционалности за обекти, които имат общи черти) и създаването на йерархия по естествен начин.
В C++ е позволено множествено наследяване, което мощен инструмент, но може да доведе до проблеми, ако не се използва внимателно. Един такъв потенциален проблем е диамантения проблем, който ще обсъдим в точка 7 от изложението.

1.4. Полиморфизъм. Полиморфизмът в C++ означава, че един и същ обект (или функция) може да се държи различно в различни сценарии. Обектите в една йерархия на наследяване споделят някакво поведение (на най-базовия клас), което може да се извика или пък да се извика предефинираното поведение на класа наследник.
В C++ полиморфизмът се осъществява по два начина:

- По време на компилация:
  - Написване на функции с едни и същи имена, но с различен брой аргументи или различни типове аргументи (но не и различни стойности на връщане);
  - Пренаписване на оператори.
- По време на изпълнение:
  - Пренаписване на функции от базовия обект в дериватния обект;
  - Виртуални функции – функции, дефинициите на които се оставят да бъдат дефинирани от обектите наследници.

2. Класове.

2.1. Дефиниция.

Класовете в обектно ориентираните езици представляват шаблон (рецепта), по който се генерират инстанции на класа. Инстнциите 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.

2.2. Дефиниране на класове.

Един клас в C++ се дефинира с помощта на ключовата дума class, последвана от името на класа и тяло, което е обградено от къдрави скоби и завършва с „;“. В тялото на един клас се съдържат декларации на член-данни и член-функции (методи), които да могат да бъдат декларирани в различни модификатори за достъп – private, protected или public. Private модификатора забранява всякакъв външен достъп до данните/методите, декларирани в него. Protected модификатора забранява външен достъп до данните/методите, декларирани в него, но те могат да бъдат достъпени от дериватните му класове (неговите наследници). Public модификатора позволява всякакъв достъп до данните/методите, декларирани в него. Когато един клас бива дефиниран, никаква памет не се заделя за този клас – няма и причина да се заделя. Това става чак тогава, когато се създават обекти – за обекти се заделя памет, но за класове - не.

Примерна декларация на клас в C++:

```
template<typename K, typename V>
class Pair {
private:
    K key;
    V value;
public:
    Pair( ) {
        key = K( );
        value = V( );
    }

    void setKey(K key) {
        this->key = key;
    }

    void setValue(V value) {
        this->value = value;
    }

    K getKey( ) {
        return key;
    }

    V getValue( ) {
        return value;
    }
};
```

Класът от примера е шаблонен и репрезентира двойка от ключ и стойност. В случая ключовата дума private може да се пропусне, тъй като в класовете по подразбиране данните и методите са private, докато при структурите са public.

Препоръчва се член-данните да се декларират в нарастващ ред по броя на байтове които заемат в паметта. Така за повечето реализации се получава оптимално изравняване до дума.

2.3. Област на класове. Класовете могат да се декларират на различни нива в програмата: глобално (на ниво някое пространство от имена) и локално (вътре във функция или в тяло 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 възмжно. Областта на локално дефиниран клас е в рамките на тялото на съответната функция или клас (тоест те са видими само там).

2.4. Обекти. Един обект представлява инстанция на даден клас. По този начин за даден клас може да имаме множество обекти. За да се използват декларираните в класа данни и методи, трябва да дефинираме негов обект. По време на самата дефиниция на обект, се заделя и паметта, нужна за съхраняването му от съответния клас. Дефиницията на обектите се осъществява подобно на скаларните типовете данни по следния начин: T object;
Където T е името на дадения клас, чийто обект искаме да дефинираме, а object е името на обекта. За да инициализираме обекта (да зададем някакви начални стойности на данните на самия обект) е нужно да използваме конструктори.
При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира на едно място в паметта. Върху какъв контекст се извиква даден метод на клас се определя от оператора this, който сочи към конкретния обект.

3. Конструктори.

3.1. Дефиниция.

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и други дейности, които съвкупно се наричат инициализация на обекта. В C++ тези дейности се изпълняват от конструкторите, които представляват специален вид член-функции. Поради това, тя се различава по някои характеристики от нормалните член-функции:

- Името на конструктора (тоест името на функцията) съвпада с името на класа;
- Типът на резултата е указател към новосъздадения обект и не се указва явно;
- Изпълнява се автоматично само при създаване на обекти.

3.2. Дефиниране на конструктори.

Дефиницията на конструктор изглежда по следния начин:

```
<class name>(<parameter list>) : <field_1>(value_1), ..., <field_n>(value_n) {
        <body – operators and definitions>
}
```

Списъкът след двоеточието представлява инициализиращ списък. Възможно е член данна да се свърже с инициализираща стойност в инициализиращия списък. Не е задължително всички член данни да бъдат инициализирани чрез този списък – възможно е някои член данни да бъдат инициализирани в тялото на конструктора, а други – в списъка. Употребата на инициализиращият списък предшества изпълнението на тялото на конструктора.
Трябва да отбележим, че е възможно един клас да не дефинира конструктори. В този случай, компилаторът автоматично ще генерира конструкор по подразбиране, който да се използва. Конструкторът по подразбиране, както ще видим по-долу, няма списък от формални параметри. Възможно е да се дефинират няколко конструктора в рамките на един клас.

3.3. Видове конструктори.

3.3.1. По подразбиране.

Генерира се автоматично от компилатора, когато не са дефинирани други конструктори. Не приема аргументи и единствено инициализира всички полета със стойностите им по подразбиране или със специфично избрани стойности, ако конструктора по подразбиране е предефиниран. Извиква се когато обект се създава без скоби, например:

```
class A {
   A( ) {
      // logic of the constructor
   }
   // other logic of the class
};
A a; // default constructor called
```

3.3.2. Параметризиран конструктор.

Конструктор, който приема параметри. Пише се от програмиста, който решава и как точно да използва подадените му като аргументи параметри, за да инициализира с тях инстанция на класа. Може да има произволен брой параметризирани конструктори. При наличието на поне един конструктор в дефиницията на класа, конструкторът по подразбиране бива предефиниран и повече не може да се използва. Ако все пак искаме да може да генерираме обекти използвайки конструктор по подразбиране, трябва да напишем и конструкторът по подразбиране, дори и тялото му да е празно.

3.3.4. Копиращ конструктор.

Това е специален параметризиран конструктор, който като единствен параметър получава псевдоним (A&) на обект от същия клас. Работата му е да клонира подадения обект. Компилаторът създава копиращ конструктор по подразбиране. При наличието на динамично заделяне на памет за член данни на класа, този конструктор трябва да бъде предефирниран ръчно от програмиста.

```
class A {
   A(const A& a) {
      // logic of the copy constructor
   }
   // other logic of the class
};
A a;
A b = A(a); // copy constructor called
```

3.3.3. Преместващ конструктор.

Приема единствен параметър от тип T& и работата му е да премести съдържанието на подадения обект в новосъздаващия се. Обектът-параметър остава празен и най-вероятно няма да се използва повече. Компилаторът създава такъв конструктор по подразбиране.

```
class A {
   std::string s;
   int k;

   A() : s("test"), k(-1) {}

   A(const A &o) : s(o.s), k(o.k) { std::cout << "move failed!\n"; }

   A(A &&o) noexcept:
         s(std::move(o.s)),       // explicit move of a member of class type
         k(std::exchange(o.k, 0)) // explicit move of a member of non-class type
   {}
};
```

4. Указатели към обекти. Масиви и обекти. Динамични обекти.

4.1. Указатели към обекти.

Подобно на атомарните променливи може да правим указатели към обекти.

```
class A {
  // class logic
};

A a;
A* ptr = &a;
ptr->class_method_name; // calling a method of the class from a pointer
```

Всеки указател към обект в паметта:

- заема константна памет: 4 байта при 32-битова машина или 8 байта при 64-битова машина;
- съхранява адреса на обекта, към когото сочи, а не самият обект;
- се дефинира като се посочи името на класа, последвано от символа * и името на указателя;
- се инициализира при самата дефиниция, като се посочи адрес на даден обект;
- може да бъде свързан с нулевия указател – nullptr;
- може да върне обекта, към който сочи, след като се дереференцира, т.е. се приложи унарния оператор *;
- може да бъде подложен на логически операнд от следните: +, – , ++ , –– , == , != , > , >= , < , <= .

Указателят this, сочещ към конкретната инстанция на класа, е указател от тип <class name>*.

4.2. Масиви от обекти.

Елементите на масив могат да бъдат и обекти, но те трябва да са от един и същи клас, тъй като масива е хомогенна линейна структура, а не хетерогенна. Единственото условие е класът да има конструктор по подразбиране, който да се извика, когато масивът бъде дефиниран.

```
class A {
  // class logic
};

A array[10]; // the default constructor is called for the 10 elements created
```

Благодарение на полиморфизма, масивите могат да бъдат и ковариантни и по този начин да са хетерогенни – ако масивът е от указатели към някакъв базов клас на дадена йерархия. По този начин този масив формално ще съдържа указатели към базов клас, но тези указатели може да сочат към обекти наследници в йерархията.

Един обект може да съдържа масив от произволен тип като своя член-данна. В този случай, тя ще може да се достъпва и инициализира като всяка друга член-данна.

4.3. Динамични обекти.

Динамичните данни са такива обекти, чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно. Динамичните обекти се съхраняват в динамичната памет (heap).

Създаването и разрушаването на динамични обекти в C++ се осъществява съответно чрез операторите new и delete. Извикването на new заделя в heap-а необходимата памет и връща указател към първия байт от структурата ѝ. Този указател може да се съхрани и да се пази, докато е необходимо. Освобождаването на паметта от heap-а става посредством оператора delete. На всяко заделяне на памет с new трябва да съответства изтриването ѝ с delete (в противен случай ще има memory leak), за да може тази памет да се използва за други цели. Това се налага, тъй като динамичната памет не се изчиства автоматично, а в C++ липсва система за самоизчистване на паметта, която е динамично заделена и няма от къде да се адресира (garbage collector).

```
class A {
  // class logic
};

A* a = new A();
A* arr = new A[10];

delete[] arr;
delete a;
```

По този начин ще се разруши обекта, адресиран от указателя. Паметта, която заема този обект се освобождава. Ако обектът, адресиран от указателя, е обект на клас, първо се извиква десктруктора на класа, а след това се освобождава паметта. За да се разруши масив, създаден чрез оператора new, трябва да се използва следният синтаксис: delete[ ] <array_name>;
Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент.
Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай, действието му е непредсказуемо. delete може да се прилага и към указател със стойност 0 (nullptr), защото той не адресира нищо и фактически не заема динамична памет.
Динамичната памет не е неограничена. Ако тя бъде изчерпана по време на изпълнение на програмата, операторът new ще върне nullptr.
Един популярен динамичен обект е динамичният масив – масив, който може да променя своята дължина. Начинът, по който той прави това е заделяне на първоначална памет с оператора new. След като този капацитет бъде запълнен до някакво ниво (например 75% или 100%), се заделя нова памет (обикновено 2 пъти по-голяма от старата), масивът се копира в новата памет, след което старата динамична памет бива изчистена чрез оператора delete.

5. Деструктори.

5.1. Дефиниция.

Разрушаването на обекти на класове в някои случаи е свързано с извършването на определени действия, които се наричат заключителни. Пример за такива действия са освобождаване на динамична памет и възстановяване на състояние на програма. Заключителните действия имат ефект, противоположен на инициализацията. За да не се специфицира всеки път една и съща последователност от заключителни действия за всеки обект на даден клас, се използва деструктора, който се извиква автоматично при разрушаването на даден обект. Това разрушаване може да настъпи поради две причини:

- Извика се оператора delete*;*
- Излиза се от блока, в който е бил създаден обект от класа.

В един клас може да присъства точно един явно дефиниран деструктор. Неговата дефиниция изглежда по следния начин: ~<class_name>( ) {...}
Дефинирането на деструктор не винаги е наложително. Член-данните на даден обект автоматично се разрушават при разрушаването на самият обект. Но, когато някои член-данни са обвързани с динамичната памет, използването на деструктор е задължително, тъй като трябва да се освободи заделената динамична памет. Ако в класът явно не е указан деструктор, то самият компилатор предоставя деструктор по подразбиране (но той не решава проблемите с динамично заделените обекти).

```
class A {
   ~A() {
      // code of the destructor
   }
};
```

5.2. Създаване и разрушаване на обекти на класове.

Съществуват два начина за създаване на обекти:

- Чрез дефиниция;
- Чрез new и delete;

В първия случай, даден обект се създава в рамките на функция или блок и се разрушава при завършването на изпълнението на функцията или излизането от блока. При създаването на обекта, може да се използва както обикновен конструктор (предефиниран или по подразбиране) така и конструктор за копиране. Разрушаването на обекта е свързано с извикването на деструктора на класа или ако такъв не е явно дефиниран, се извиква генерирания от компилатора деструктор.

Във втория случай създаването и разрушаването на обектите се управлява от програмиста. Той създава обекти чрез оператора new и ги разрушава чрез оператора delete. Употребата на new се включва в конструкторите, а на delete – в деструктора на класа.

```
class A {
   int *arr;
public:
   A() {
      arr = new int[10];
   }

   ~A() {
      delete[] arr;
   }
};
```

6. Оператори.

6.1. Дефиниция.

Операторите са конструкции в програмните езици, които репрезентират някакво поведение на обекта, към който се прилагат, също като функциите, но с друг синтаксис.

```
class A {
   // definition
} a;

int A[] = new A[2]{1, 2};
cout << a[0] + a[1];
```

В примера по-горе използваме три оператора (в червено): оператор за принтинаране, оператор за индексиране и оператор за събиране.

6.2. Предефиниране на оператори.

Атомарните типове предоставени от програмния език C++ имат дефинирани оператори (за сравнение, за принтиране, за събиране и т.н.). Обектите на класове, които са създадени от програмиста обаче – нямат и е добра практика програмиста да предефинира операторите за тях (или поне тези, които може да се използват). Предефинираните оператори могат да се държат по произволен начин, но е добра практика да имат сходно на стандартните оператори поведение, но за обектите от персонализирания клас.

```
class A {
   int a;
public:
   A operator+(const A& other) const {
      return this->a + other.a;
   }
};
```

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;
- приоритет;
- асоциативност.

Позицията на оператора спрямо аргументите му го определя като:
- Префиксен (операторът стои пред единсвтвения аргумент, към който се прилага);
- Инфиксен (операторът стои между двата аргумента, към които се прилага);
- Постфиксен (операторът стои след единствения аргумент, към който се прилага).

Приоритетът определя реда на изпълнение на операторите в някакъв израз (редица) от оператори. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет. Например приоритета на умножението и делението е по-висок от приоритета на събирането и изваждането.

Асоциативността определя реда на изпълнение на операторите с еднакви приоритети в даден израз. В C++ има ляво асоциативни и дясно асоциативни оператори. Първите оператори се изпълняват отляво надясно, а вторите от дясно на ляво.

В C++, оператори като ::, ?:, ., *, # и ## не могат да бъдат предефинирани.

Когато предефинирането на оператор изисква достъп до компонентите на класа, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел (friend functions) на тези класове. Функциите-приятел на даден клас имат достъп до всички негови компоненти и се отбелязват със запазената дума friend пред декларацията си. Функциите-приятели се дефинират в public частта на класа.

Предефинирането на бинарните оператори изисква дефинициите да са членове на класа на левия си аргумент или да са свободни функции. Следователно не всички оператори могат да се предефинират като член функции на класа си. Такива например са << и >> (bitshift операторите, които се използват и за входно изходни операции). Те се предефинират чрез приятелски функции, които са вид свободни функции и имат достъп до всички данни и методи на класовете, чиите приятели са.

```
class A {
    int a;
public:
    friend ostream& operator<<(ostream& os, const A& a) { os << x.a; return os; }
};
```

7. Производни класове.

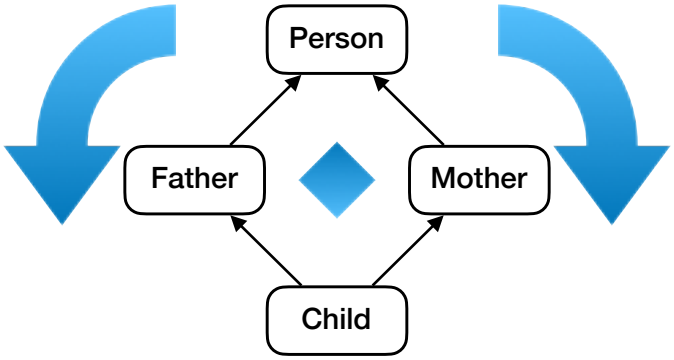
7.1. Дефиниция.

Наследяването е механизъм, чрез който дериватния (производния) клас бива базиран върху логиката на друг базов клас. Дериватния клас наследява, разширява и предефинира интерфейса на базовия клас.

```
class A {
public:
    void method( ) {
        // method logic
    }
};
class B: protected A {
public:
    void method( ) {
        // method logic
    }
};
B b;
b.method( );
```

Както споменахме, в C++ е позволено множествено наследяване. Дефинираме го по следния начин: class A: public B, private C. В този случай при създаването на нов обект от клас A, първо ще се извика конструктура на базовия клас B, след това конструктора на базовия клас C и накрая ще се изпълни конструктора на дериватния клас A. При деструкторите извикването е в обратния ред: C, B, A.

Диамантеният проблем възниква, когато дериватен клас наследява от два родителски класа, които споделят общ базов клас. Това е илюстрирано с диаграмата по-долу:



Тук имаме клас Child, наследяващ класовете Father и Mother. Тези два класа, от своя страна, наследяват класа Person, защото и бащата, и майката са Person.

Както е показано на фигурата, клас Child наследява чертите на клас Person два пъти – веднъж от бащата и отново от майката. Това поражда двусмислие, тъй като компилаторът не успява да разбере кой път да поеме.

```
class Person { //class Person
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
};

class Father : public Person { //class Father inherits Person
public:
    Father(int x):Person(x) { cout << "Father::Father(int) called" << endl; }
};

class Mother : public Person { //class Mother inherits Person
public:
    Mother(int x):Person(x) { cout << "Mother::Mother(int) called" << endl; }
};

class Child : public Father, public Mother  { //Child inherits Father and Mother
public:
    Child(int x):Mother(x), Father(x) { cout << "Child::Child(int) called" << endl; }
};

int main() {
    Child child(30);
}
```

Изходът от програмата ще е следния:

```
Person::Person(int) called
Father::Father(int) called
Person::Person(int) called
Mother::Mother(int) called
Child::Child(int) called
```

Конструкторът на клас Person се извиква два пъти: веднъж, когато се създава обектът на класа Father и след това, когато се създава обектът на класа Mother. Свойствата на класа Person се наследяват два пъти, което води до неяснота.

Тъй като конструкторът на клас Person се извиква два пъти, деструкторът също ще бъде извикан два пъти, когато обектът на клас Child бъде унищожен.

Решението на проблема с диаманта е използването на ключовата дума virtual. Ние правим двата родителски класа (които наследяват от един и същи родителски клас) във виртуални класове, за да избегнем две копия на родителския клас в дъщерния клас. Виртуалното наследяване гарантира, че се предава само един екземпляр от наследения клас (в този случай класът Person). С други думи, класът Child ще има един екземпляр на класа Person, споделян от класовете Father и Mother. Чрез наличието на един екземпляр на класа Person двусмислието се разрешава.

Едно нещо, което трябва да се отбележи относно виртуалното наследяване е, че дори ако параметризираният конструктор на класа Person е изрично извикан от конструкторите на класове Father и Mother чрез списъци за инициализация, ще бъде извикан само основният конструктор на класа Person.

7.2. Наследяване и достъп до наследените компоненти.

При наследяване на базов клас, дериватния клас има достъп до public и protected полетата и методите на базовия клас. Всички данни на базовия клас, които са private остават скрити за дериватния клас.

В C++ в зависимост от типа на наследяването, тези наследени член-данни и методи се третират по различен начин.

модификатор за достъп	модификато на базов клас	Наследява се като
public	private protected public	private protected public
protected	private protected public	private protected protected
private	private protected public	private private private

Наследяването по подразбиране е private.