Обектно-ориентирано програмиране с C++
Пълен сборник (ТЕМА 1 – ТЕМА 26)

Документът съдържа подробно обяснение на всички теми, кодови примери и концепции, оформени за удобно четене и подготовка за изпит.

ТЕМА 1: Обекти и класове. Дефиниция на клас. Общи понятия и концепции.

1. Исторически контекст и същност на ООП Езикът C++ (създаден през 1985 г.) надгражда процедурния език C (1972 г.), като добавя поддръжка за Обектно-ориентираното програмиране (ООП). Всъщност, C++ е толкова близък до предшественика си, че при компилация често кодът се превежда първо до C. Основната сила на ООП е в подхода към решаването на проблеми. Вместо да пишете дълъг списък от инструкции, ООП позволява разбиването на сложния проблем на по-малки, самостоятелни части, наречени обекти. Всеки обект е капсулирана единица, която съдържа свои собствени данни и инструкциите за работа с тях.

Всички ООП езици стъпват на три фундаментални принципа (стълба):

Капсулиране: Това е защитният механизъм. Той свързва кода и данните в едно цяло и ги предпазва от външна намеса и неправилна употреба. Мислете за това като за "черна кутия" – вие ползвате функционалността, без да виждате сложните механизми вътре.

Полиморфизъм (Многоформено): Това е способността на група от различни обекти да бъдат третирани като един общ тип. Той позволява едно и също име (на функция или оператор) да се използва за различни цели в зависимост от контекста.

Наследяване: Механизъм, при който един обект придобива свойствата на друг. Това създава връзката "is a" (е вид). Например, "Кучето" е вид "Животно" и наследява неговите характеристики. C++ е специфичен с това, че позволява множествено наследяване – един клас може да има повече от един родител.

2. Разлика между Клас и Обект Това е най-важното разграничение в тази тема.

Класът е логическа абстракция и дефиниция на нов тип данни. Той описва как ще изглеждат обектите, но самият той не заема памет (освен за статични данни). Класът е чертежът или шаблонът. Задължително е класът да има име, което се превръща в име на типа.

Обектът е физическата инстанция на класа. Когато дефинирате обект, той вече заема памет (обикновено в Heap/динамичната памет за по-сложните обекти). Обектът е обединението на конкретните данни и функциите, които ги обработват.

3. Видове класове В теорията и практиката на C++ разграничаваме няколко специфични вида класове:

Абстрактен клас: Това е клас, който не е предназначен да създава обекти, а да служи само като база (родител) за други класове. Той задължително съдържа поне една "чисто виртуална функция" (pure virtual function), която се бележи със синтаксиса = 0.

Статичен клас (Singleton концепция): В езици като C# има static class, който не може да се инстанцира. В C++ това се постига чрез архитектурни модели, при които не се позволява създаването на повече от един обект от този клас.

Примерен код (Дефиниция на клас и обект):

```
#include <iostream>
using namespace std;

// Дефиниция на КЛАС (Шаблонът)
class Person {
public: // Модификатор за достъп
    int age; // Даннов член
};

int main() {
    // Създаване на ОБЕКТ (Инстанцията)
    // Person е типът, personObj1 е променливата/обекта
    Person personObj1;
    // Достъп до член на обекта чрез оператор точка (.)
```

```
    personObj1.age = 21;

    cout << personObj1.age;
    return 0;
}
```

ТЕМА 2: Методи и параметри. Даннови членове и пропъртита.

1. Структура на класа Вътрешността на класа се състои от два основни компонента, които работят заедно:

Даннови членове (Fields/Properties): Това са променливите, декларирани вътре в класа. Те съхраняват състоянието на обекта (напр. age, name). Всеки обект пази свое собствено копие на тези данни (освен ако не са static).

Методи (Member Functions): Това са функциите, които принадлежат на класа. Те дефинират поведението и манипулират данните. Методите приемат параметри (входни данни), за да извършат своята работа.

2. Дефиниране на методи В C++ имате гъвкавост къде да напишете кода на метода:

Вътрешно (Inline): Кодът се пише директно в тялото на класа. Подходящо за кратки функции.

Външно: В класа се пише само декларацията (прототипа), а самата реализация се изнася извън него, използвайки оператора за обхват :: (напр. Person::run()). Това прави кода по-чист и четим при големи проекти.

3. Жизнен цикъл: Конструктори и Деструктори Това са специални методи, които управляват раждането и смъртта на обекта.

Конструктор: Винаги носи същото име като класа и няма тип на връщан резултат (дори не е void). Неговата задача е инициализация – да зададе начални стойности на данните, за да не е обектът в невалидно състояние. Извиква се автоматично при създаването на обекта. Може да имате няколко конструктора с различни параметри (сигнатури).

Деструктор: Името му е името на класа с тилда (~) отпред. Извиква се автоматично, когато обектът се унищожава (излиза от обхват или се изтрива с delete). Неговата роля е да освободи заетата памет. Добра практика е деструкторът да бъде виртуален (virtual), за да се гарантира правилното унищожаване при наследяване.

Примерен код (Методи и Конструктор):

```
class Person {
public:
    int age;

// Конструктор (инициализира възрастта)
    Person(int initialAge) {
        this->age = initialAge;
    }

    // Метод (дефиниран вътре)
    void printAge() {
        // this-> сочи към текущия обект
        cout << "Age: " << this->age << endl;
    }

    // Декларация на метод (ще се дефинира външно)
    void sleep();
};

// Външна дефиниция
void Person::sleep() {
    // Implementation
}
```

ТЕМА 3: Модификатори на достъп в клас.

Модификаторите за достъп са "ключалките" на вашата програма. Те реализират принципа на капсулиране, като определят кои части от кода са видими за външния свят и кои са скрити. Това предотвратява грешки, породени от неправилна употреба на вътрешните данни.

1. Основни модификатори (C++ и C#)

Public (Публичен): Най-отвореното ниво. Членовете, маркирани като public, са достъпни от всяка точка на програмата, която има достъп до обекта. Обикновено методите са публични, за да образуват "интерфейса" на класа.

Private (Частен): Най-строгото ниво. Членовете са достъпни само от методите на същия клас. Никой друг обект или външна функция не може да ги вижда. Данновите членове почти винаги трябва да са private.

Protected (Защитен): Това е специален баланс между public и private, предназначен за йерархии от класове. Protected членовете са скрити за външния свят (като private), но са напълно достъпни за класовете-наследници (derived classes).

2. Приятелски класове (Friend) – Специфика на C++ C++ предлага механизъм за "заобикаляне" на правилата за достъп чрез ключовата дума friend. Ако клас A обяви клас B за свой "приятел", то клас B получава пълен достъп до всички private и protected членове на A.

Важно: Приятелството не е двупосочно (ако аз съм ти приятел, ти не си автоматично мой) и не се наследява. Това трябва да се използва внимателно, защото нарушава енкапсулацията.

Примерен код (Modificators & Friend):

```
class MyClass {
private:
    int privateVar; // Скрито от света
```

public: MyClass(int val) : privateVar(val) {} // Публичен конструктор

```
// Обявяваме FriendClass за приятел
    friend class FriendClass;
};

class FriendClass {
public:
    void AccessPrivateVar(MyClass& obj) {
        // Тук имаме достъп до privateVar, защото сме "friend"
        std::cout << obj.privateVar << std::endl;
    }
};
```

ТЕМА 4: Accessor и Mutator методи.

Когато направим данните private (за да ги защитим), ние все пак трябва да предоставим начин на останалата част от програмата да работи с тях, но контролирано. Тук на помощ идват Accessor и Mutator методите, често наричани Getters и Setters.

1. Mutator методи (Setters)

Цел: Използват се за промяна (задаване) на стойността на private променлива.

Предимство: Те позволяват валидация. Вместо директно да присвоите age = -5 (което е невалидно), вие викате setAge(-5). В метода setAge може да има if проверка, която да отхвърли отрицателната стойност и да запази обекта валиден.

2. Accessor методи (Getters)

Цел: Използват се за четене (връщане) на стойността на private променлива.

Предимство: Позволяват да направите променливата "Read-only" (само за четене), като предоставите Getter, но не и Setter. В C++ тези методи често се маркират като const (напр. int getAge() const), което гарантира на компилатора, че извикването им няма да промени състоянието на обекта.

3. Свойства (Properties) в C# В езика C# тази концепция е вградена синтактично чрез т.нар. Properties. Те изглеждат като променливи за външния свят, но отзад стоят get и set блокове код. В C++ това се симулира чрез отделни методи.

Примерен код (Getter и Setter в C++):

```
class Person {
private:
    string name; // Private данна
```

public: // MUTATOR (Setter) void setName(const string& newName) { // Тук можем да добавим проверка за празен низ this->name = newName; }

// ACCESSOR (Getter) string getName() const { return this->name; } };

```
int main() {
    Person person;
    person.setName("Larry"); // Записване чрез метод
    // Директен достъп person.name = "Larry" би дал ГРЕШКА
    cout << person.getName(); // Четене чрез метод
}
```

ТЕМА 5: Методи на клас. Видове и модификатори. Припокриване.

Методите са двигателят на класа. Те не са просто функции, а инструменти за моделиране на поведение.

1. Видове методи според връзката им с обекта

Instance Methods (Методи на член): Това са стандартните методи. Те работят върху конкретен обект и имат достъп до неговите данни чрез скрития указател this.

Static Methods (Статични методи): Тези методи са асоциирани със самия клас, а не с конкретна инстанция (обект). Те могат да се извикват директно чрез името на класа (напр. Math::Sqrt()) и нямат достъп до не-статични данни, защото не знаят "кой" е обектът.

Конструктори и Деструктори: Специализирани методи за управление на паметта и инициализацията.

2. Припокриване на методи (Method Overloading) Това е мощен механизъм, който позволява в един клас да имаме няколко метода с едно и също име.

Как работи: Компилаторът ги различава по техните параметри (брой или тип). Това се нарича "сигнатура" на метода.

Пример: Можем да имаме метод Print(int x) и друг метод Print(string s). Програмата автоматично ще избере правилния вариант в зависимост от това какво подадем.

3. Припокриване на оператори (Operator Overloading) В C++ можем да накараме нашите обекти да работят със стандартни оператори като +, -, ==. Това прави кода много по-четим. Вместо да пишем obj1.add(obj2), можем да напишем obj1 + obj2, ако сме дефинирали специален метод operator+.

Примерен код (Overloading):

```
class MyClass {
public:
    // Вариант 1: Приема цяло число
    void Process(int x) {
```

```
        cout << "Processing integer: " << x << endl;
    }
// Вариант 2: Приема две цели числа (Overloading)
    void Process(int x, int y) {
        cout << "Processing two integers: " << x << ", " << y << endl;
    }

    // Вариант 3: Приема дробно число
    void Process(double a) {
        cout << "Processing double: " << a << endl;
    }
};
```

int main() { MyClass obj; obj.Process(42); // Извиква Вариант 1 obj.Process(10, 20); // Извиква Вариант 2 obj.Process(3.14); // Извиква Вариант 3 }

ТЕМА 8: Структуриране на класова йерархия

1. Същност на наследяването и йерархията В Обектно-ориентираното програмиране (ООП) класът е шаблон, който дефинира вида и поведението на обекта. Често обаче обектите в реалния свят споделят общи характеристики. Например, и колата, и камионът са "превозни средства". И буквата, и цифрата са "символи". За да не пишем един и същи код многократно, използваме Наследяване.

Терминология: Класът, който дава своите характеристики, се нарича Родител (Parent, Superclass), а класът, който ги приема, се нарича Дете (Child, Subclass).

Връзката "Is-a": Наследяването създава връзка от типа "е вид". Цифрата е вид Символ.

Механизъм: Когато детето наследява родителя, то получава достъп до неговите полета (променливи) и методи (функции), стига те да са с права за достъп public или protected.

2. Синтаксис и Модификатори на наследяване в В наследяването се описва чрез следния синтаксис: class ChildName : AccessMode ParentName. Ключовият момент тук е AccessMode (начинът на наследяване). Той променя видимостта на членовете, които идват от родителя, когато влязат в детето:

Public наследяване: Най-често използваното. Всичко public от родителя остава public в детето. Всичко protected остава protected.

Protected наследяване: Всички public и protected членове на родителя се превръщат в protected в детето.

Private наследяване: Всички public и protected членове на родителя стават private в детето. Ако не напишете нищо, това е режимът по подразбиране за класове.

Пример за влиянието на модификаторите:

```
class A {
public: int x;
protected: int y;
private: int z;
};

// При public наследяване: x остава public, y остава protected
class B : public A {};

// При private наследяване: x и y стават private за D
class D : private A {};
```

3. Видове наследяване В зависимост от броя на родителите и структурата на връзките, различаваме пет основни вида:

Единично наследяване (Single Inheritance): Детето има само един родител. Това е най-простата форма (Car наследява Vehicle).

Множествено наследяване (Multiple Inheritance): Детето има повече от един родител. позволява това. Например, клас Car може да наследи едновременно Vehicle и FourWheels. Синтаксисът използва запетая:

class Car : public Vehicle, public FourWheels.

Наследяване на слоеве (Multilevel Inheritance): Детето наследява от клас, който вече е дете на друг клас (Верига: Дядо -> Баща -> Дете). Например Vehicle -> FourWheels -> Car.

Наследяване чрез йерархия (Hierarchical Inheritance): Един родител има множество деца. Например Vehicle е родител едновременно на Bus и на Car.

Смесено/Виртуално наследяване (Hybrid Inheritance): Комбинация от горните видове. Например комбинация от йерархично и множествено наследяване.

4. Полиморфизъм и разширяване на поведението Целта на наследяването не е само копиране на код, а разширяване и специализиране на поведението.

Предефиниране (Overriding): Детето може да промени поведението на метод, наследен от родителя. Например, ако родителят има метод draw(), детето може да го пренапише, за да рисува нещо специфично.

Upcasting: Когато извикате метод върху обект от йерархията, инструкцията "тръгва" от най-ниското ниво (детето) и се изкачва нагоре (upcasting), докато намери първата реализация на метода. Ако детето е пренаписало метода, ще се изпълни неговата версия – това е същността на полиморфизма.

ТЕМА 9: Оценка на качеството на кода. Свързаност и структурираност. UML диаграми.

За да се оцени качеството на една архитектура, често се използва визуализация чрез UML (Unified Modeling Language). Класовата диаграма показва статичната структура и връзките между класовете.

1. Връзки между класовете в UML Освен наследяването, обектите си взаимодействат по други начини. Важно е да разграничавате следните видове връзки:

Асоциация (Association): Връзка "Клас X познава Клас Y". Обикновено това означава, че Клас X съдържа указател (pointer) към Клас Y.

Зависимост (Dependency): Връзка "Клас X използва Клас Y". Тук връзката е по-слаба – Клас Y се появява само като локална променлива в метод на X или като аргумент на функция, но не е постоянна част от X.

Съвкупност (Aggregation): Връзка "Клас X притежава Клас Y", но е слаба връзка (Контейнер).

Ключова характеристика: Жизненият цикъл на съдържанието НЕ зависи от контейнера.

Пример: Група (контейнер) и Студент (съдържание). Ако закрием групата, студентът продължава да съществува.

Композиция (Composition): Връзка "Клас X притежава Клас Y", но е силна връзка.

Ключова характеристика: Жизненият цикъл на съдържанието ЗАВИСИ от контейнера. При унищожаване на контейнера, умира и съдържанието.

Пример: Къща (контейнер) и Спалня (съдържание). Ако разрушим къщата, спалнята престава да съществува.

2. Метрики за качество: Coupling и Cohesion Два основни термина дефинират дали кодът е добре написан:

Еднородност (Cohesion): Определя колко фокусиран е един клас върху задачите си.

Нисък коефициент: Класът прави твърде много различни неща (лошо). Пример: Клас Staff, който проверява имейли, праща писма, валидира данни и печата.

Висок коефициент: Класът е тясно специализиран (добро). Пример: Клас Staff, който се грижи само за заплатата и данните на служителя, а действията с имейл са изнесени другаде.

Свързаност (Coupling): Определя колко зависими са класовете един от друг.

Висока свързаност (High Coupling): Класовете са силно "заплетени". Промяна в един клас чупи логиката в други. Това прави кода труден за поддръжка.

Ниска свързаност (Low Coupling): Класовете са независими. Промяна в един не влияе на другите.

Златно правило: Качествената програма има Висока Еднородност (High Cohesion) и Ниска Свързаност (Low Coupling).

ТЕМА 10: Качество на код: дублиращи се фрагменти. Целево-ориентиран проект.

1. Проблемът с дублиращия се код (Code Duplication) При разрастване на проектите, често се налага един и същи код да се ползва на много места.

Пример: В една игра постоянно трябва да се рисува графичен елемент (кораб). Това изисква сложен код за работа с пиксели и координати.

Грешният подход: Ако копираме и поставяме (copy-paste) този сложен код навсякъде, където корабът се движи, създаваме "дублиращи се фрагменти". Това прави програмата трудна за четене и поддръжка.

Правилният подход (Стандартизация): Дублиращият се код трябва да се изнесе в отделна функция (метод). Така, ако решим да сменим цвета на кораба от червен на син, го променяме само на едно място (във функцията), вместо да търсим стотици копия на кода.

Пример за рефакториране:

// Сложно (Дублиране): circle(10, 10, 5); fillCircle(10, 10, Red);

// Разбираемо (Функция): displayShip(10, 10, 5, Red); // Изнасяме логиката тук

2. Именуване и Коментиране Качеството на кода зависи и от четимостта.

Имена: Името на функцията трябва ясно да казва какво прави тя. При големи проекти програмистите прекарват повече време в четене, отколкото в писане.

Коментари: Добрият коментар не обяснява синтаксиса (какво става), а логиката (защо става).

Лош коментар: int i=5; // създаваме променлива i равна на 5. Това е излишно за програмист.

Добър коментар: Обяснява входните параметри, какво връща функцията, граничните стойности и странните решения. Примерно: "Ако number е четно, връща 1. При отрицателно число връща 0". Това позволява на друг програмист да разбере функцията, без да чете целия ѝ код.
ТЕМА 11: Проектиране на обектно-структуриран код1. Необходимост от структурираненСофтуерът никога не е статичен продукт. Той не се прави веднъж завинаги, а постоянно се променя, допълва и приспособява, често от различни програмисти1. Поради тази динамика, кодът трябва да бъде обектно-структуриран и добре описан. Това гарантира, че всеки, който го погледне, ще разбере лесно за какво става въпрос2.2. Метрики за качество на кодаДва основни фактора определят качеството на архитектурата:Свързаност (Coupling):Представлява връзката между отделните единици (класове) в програмата3.Плътна свързаност: Ако два класа споделят твърде много общи характеристики или зависят силно един от друг4.Цел: В добре структуриран код свързаността трябва да се избягва (минимизира)5.Решение: Постига се чрез наследяване и абстракция. Това позволява промяна на един клас без странични ефекти върху другите6.Еднородност (Cohesion):Отнася се до броя и еднообразието на задачите, за които една единица (клас или метод) е отговорна7.Висока еднородност: Когато единицата отговаря за само една логическа задача8.Приложение: Прилага се както върху класовете (описват един логически обект), така и върху методите (вършат една добре описана задача)9999.Полза: Използването

на описателни имена прави кода четим и ясен10.3. Въпроси при проектиранеЗа да постигнем добра структура, трябва да си задаваме следните въпроси по време на разработка11:В кой клас е най-логично да добавим този метод? 12Твърде голям ли е методът? (Ако върши повече от една задача, трябва да се разбие) 13.Кой има право да достъпва данните на класа? (Енкапсулация) 14.Твърде сложен ли е класът? (Ако описва повече от един логически обект, трябва да се раздели) 15.Локализиране на промяната: При промяна на кода, тя трябва да засегне възможно най-малко класове16.4. Пример за добра структура: Stock Trader (Търговия с акции)Примерът демонстрира разделение на отговорностите между класове Stock (Акция) и StockPurchase (Покупка).Клас Stock (Акция): Отговаря само за данните на самата акция (символ и цена).class Stock {
private:
    char symbol[SYMBOL_SIZE];
    double sharePrice;
public:
    // Конструктор по подразбиране
    Stock() { set("", 0.0); }
    // Параметризиран конструктор
    Stock(const char *sym, double price) { set(sym, price); }

// Mutator (Setter)
    void set(const char *sym, double price) { ... }
    // Accessors (Getters)
    const char *getSymbol() const { return symbol; }
    double getSharePrice() const { return sharePrice; }
};
17Клас StockPurchase (Покупка): Използва Stock, но отговаря за транзакцията (брой акции и цена на сделката).#include "Stock.h"
class StockPurchase {
private:
    Stock stock; // Композиция/Агрегация
    int shares;
public:
    // Конструктор, приемащ обект Stock
    StockPurchase(const Stock &stockObject, int numShares) {
        stock = stockObject;
        shares = numShares;
    }
    // Изчислява цена
    double getCost() const {
        return shares * stock.getSharePrice();
    }
};
18ТЕМА 12: Класове и обекти. Декларация, дефиниция, структура.1. Дефиниция на КласКласът е потребителски дефиниран тип – шаблон с полета и методи, по който се създават обекти19.Структура на дефиницията:Заглавна част: Ключова дума class и името (идентификатор)20.Тяло: Затворено във фигурни скоби { ... }21.Край: Задължително завършва с точка и запетая ;22.Класът се счита за напълно дефиниран, когато се достигне затварящата скоба на тялото. Тогава всички негови членове са известни23.2. Декларация (Forward Declaration)Възможно е клас да се декларира, без да се дефинира.Синтаксис: class Point;24.Ограничения: Когато класът е само деклариран, употребата му е силно ограничена (може да се ползват указатели към него, но не и да се създават обекти или достъпват членове)25.3. ОбектиДекларация: Създаването на променлива от типа на класа се нарича инстанциране26.Point p; (в стека)27.Point *ptr = &p; (указател)28.Инициализация: Задаване на конкретни стойности на полетата, често чрез методи29.4. Разлика между Структура (Struct) и Клас (Class)В контекста на предоставения материал се прави разграничение (характерно за C# и някои аспекти на):ХарактеристикаСтруктура (Structure)Клас (Class)ТипПримитивен/Стойностен тип (Value type) 30Референтен тип (Reference type) 31ПараметриПодават се чрез копие 32Подават се чрез референция 33НаследяванеНе поддържат наследяване 34Поддържат наследяване 35СъдържаниеИма само полета 36Има полета и методи 37Достъп (Default)Public (по подразбиране) 38Private (по подразбиране) 39ПаметСъздават се в Стека (Stack) 40Създават се в Heap (динамичната памет) 41ТЕМА 13: Конструктори и деструктори. Видове.1. КонструкторИСпециален метод за инициализация на обекта42.Характеристики:Има същото име като класа43.Няма тип на връщан резултат (дори void)44.Може да бъде презаписван (Overloading) – класът може да има много конструктори45.2. Видове конструкторИКонструктор по подразбиране (Default Constructor):Няма параметри и тяло. Ако не напишем никакъв конструктор, компилаторът автоматично създава такъв: Point();46.Важно: Ако дефинираме друг конструктор (напр. с параметри), компилаторът спира да създава служебния default конструктор. Тогава трябва ние да го напишем ръчно, ако ни трябва47.Параметризиран конструктор:Приема аргументи, за да инициализира обекта с конкретни данни още при създаването му48.Point::Point(int corX, int corY) {
    x = corX;
    y = corY;
}

49Copy конструктор:Инициализира нов обект, като копира данните от друг съществуващ обект от същия тип50.Point(const Point &p2) {
    x = p2.x;
    y = p2.y;
}
51Conversion конструктор (За преобразуване):Конструктор, който може да бъде извикан с единичен аргумент. Това позволява автоматично преобразуване на тип (напр. от int в Test)52.Пример: t = 30; ще извика Test(int i) скрито53.Explicit конструктор:Дефинира се с ключовата дума explicit. Това забранява скритото преобразуване чрез единичен аргумент54.explicit A(int); // Не може да се извика като A obj = 5;
553. ДеструкториИзползват се за унищожаване на обекта и освобождаване на паметта56565656.Синтаксис: Името на класа със знак ~ отпред (напр. ~Point())57.Нямат параметри58.В един клас може да има само един деструктор59.Извиква се автоматично в края на обхвата/програмата60.4. Методи на клас (Член-функции)Те реализират операциите върху обектите61.Правило: Всички операции върху данните (член-променливите) трябва да минават през методите на класа (енкапсулация)62.Дефинират се с тип на връщан резултат и могат да имат параметри63636363.// Дефиниция извън класа
void Point::get_x() {
    return x;
}
64
ТЕМА 14: Дефиниране на връзки. Взаимодействия на обекти по вертикала и хоризонтала.
1. Типове взаимодействия В обектно-ориентираното програмиране (ООП) обектите не съществуват изолирано. Те си взаимодействат, за да формират работеща система.

Взаимодействия по хоризонтала: Отнасят се до комуникацията между обекти от еднакъв или подобен ранг, или обекти, които си сътрудничат за изпълнение на задача. Това обикновено са връзките Association, Aggregation и Composition.

Взаимодействия по вертикала: Отнасят се до комуникацията между обекти на различни нива в йерархията. Това е класическото наследяване (Inheritance).

2. Основни видове връзки (Relationships) Връзките описват "кой какво прави" с другия. Основните три вида, подредени по сила на връзката (от най-слаба към най-силна), са:

Асоциация (Association) – "Uses-a" (Използва):

Същност: Връзка, при която обектите са независими, но знаят един за друг и си обменят информация. Връзката може да е "един към един", "един към много" и т.н..

Пример: Студент и Курс. Студентът се записва в курс, но курсът и студентът са напълно отделни същности.

Код: Реализира се чрез методи, които приемат другия обект като параметър или чрез указатели/референции.

Агрегация (Aggregation) – "Has-a" (Има):

Същност: По-слаба форма на притежание. Един обект (цялото) съдържа друг обект (частта), но частта може да съществува самостоятелно и да бъде споделяна .

Ключова характеристика: Жизнените цикли са независими. Ако унищожим контейнера, частта остава жива.

Пример: Библиотека и Книга. Ако библиотеката затвори (бъде унищожена), книгите продължават да съществуват и могат да отидат в друга библиотека.

Композиция (Composition) – "Part-of" (Част от):

Същност: Най-силната връзка. Един обект е неразделна част от друг.

Ключова характеристика: Силна зависимост на жизнения цикъл. Частта се създава и унищожава заедно с цялото. Тя не може да съществува извън него .

Пример: Автомобил и Двигател. Двигателят е създаден конкретно за тази кола като част от нея. Ако колата се пресова (унищожи), двигателят също си заминава (в контекста на модела).

Сравнителна таблица: | Характеристика | Композиция (Composition) | Агрегация (Aggregation) | Асоциация (Association) | | :--- | :--- | :--- | | Тип връзка | Част от (Part-of) | Има (Has-a) | Използва (Uses-a) | | Жизнен цикъл | Управлява се от родителя (Силна връзка) | Независим (Слаба връзка) | Независим | | Споделяне | Частта не се споделя | Частта може да се споделя | Може да се споделя |

Пример за Композиция (C++):

```
class Engine { /*...*/ };

class Car {
private:
    Engine engine; // Engine е част от Car
public:
    Car(int power) : engine(power) {} // Създава се заедно с колата
    // Когато Car се унищожи, engine също се унищожава
};
```

ТЕМА 15: Подтипове, подкласове и присвоявания. Предаване на параметри.

1. Подклас срещу Подтип (Subclass vs Subtype) Макар често да се ползват като синоними, има тънка разлика:

Подклас (Subclass): Това е техническото понятие при наследяване. Клас, който произлиза от друг (Родител) и наследява неговите атрибути и методи. Фокусът е върху повторното използване на код.

Подтип (Subtype): Това е логическото понятие, свързано с полиморфизма (Принцип на заместването). Тип данни е подтип на друг, ако може да го замести във всяка ситуация, без да наруши коректността на програмата.

Пример: Ако имаме функция, която очаква Animal, и подадем Dog, това работи, защото Dog е подтип на Animal.

2. Предаване на параметри Начинът, по който подаваме данни на функциите, е критичен за производителността и коректността.

Предаване по стойност (Pass-by-value):

Механизъм: Създава се копие на аргумента. Промените във функцията не засягат оригинала.

Проблем: При големи обекти (класове, вектори) копирането е бавно и излишно хаби памет.

Предаване по референция (Pass-by-reference):

Механизъм: Не се прави копие. Параметърът е псевдоним (alias) на оригиналния обект. Всички промени се отразяват директно върху оригинала.

Предимство: Ефективност при големи структури. Позволява функцията да върне резултат чрез промяна на аргументите .

Синтаксис C++: void func(int &ref;);.

Предаване чрез const reference (Pass-by-const-reference):

Механизъм: Подаваме референция (за бързина), но я маркираме като const (за безопасност).

Цел: Гарантираме на извикващия, че функцията няма да промени обекта му, но избягваме цената на копирането .

Синтаксис C++: void func(const int &value;);.

Сравнение с C#:

В C# обектите (reference types) по подразбиране се предават "по референция" като стойност (промените по полетата се виждат, но пренасочване на указателя - не). За истинско pass-by-reference се ползва ключовата дума ref.

ТЕМА 16: Наследяемост. Полиморфизъм. Достъп до методи и данни на различни нива.

1. Наследяемост (Inheritance) Това е процесът на моделиране на връзката "is-a" (е вид).

Конструиране: Когато създаваме обект от производен клас (Дете), редът е:

Конструира се Базовият клас (Родителят).

Инициализират се членовете на детето.

Изпълнява се тялото на конструктора на детето.

Унищожаване: Редът е обратен – от най-производния към най-базовия клас (Дете -> Родител).

2. Полиморфизъм (Polymorphism) Способността на обекти от различни класове да реагира по различен начин на едно и също извикване.

Статичен (Compile-time): Реализира се чрез Overloading (претоварване) на функции и оператори. Решава се по време на компилация.

Динамичен (Runtime): Реализира се чрез виртуални функции. Решава се по време на изпълнение чрез указатели или референции към базовия клас. В се изисква ключовата дума virtual. В C# всички методи са виртуални само ако са маркирани, а предефинирането става с override.

3. Спецификатори за достъп (Access Specifiers) Те контролират видимостта на членовете при наследяване и използване.

Спецификатор C++ Описание C# Описание public Достъпен отвсякъде.

Достъпен отвсякъде.

protected Достъпен от класа и неговите подкласове.

Достъпен от класа и неговите подкласове.

private Достъпен само от дефиниращия клас.

Достъпен само от дефиниращия клас.

friend (Само в C++) Клас или функция, изрично обявени като "приятел", имат пълен достъп до private членове.

Няма еквивалент (използва се internal за ниво проект).

Експортиране в Таблици

Забележка: При наследяване в , спецификаторът в декларацията на класа (class Derived : public Base) определя как се "превеждат" правата на наследените членове в новия клас. При public наследяване правата се запазват .
ТЕМА 20: Конструиране и деструкция на вградени обекти.
1. Йерархия на влагане (Composition) Когато един клас (Контейнер) съдържа обекти от друг клас (Вграден/Член), се създава структура на влагане. В това често се нарича "Composed class".

Правила за достъп: Вграденият клас може да бъде public, private или protected член на контейнера. Тези спецификатори имат стандартното си значение – определят кой може да вижда вградения обект .

2. Ред на конструиране Конструирането на сложен обект не е хаотично. То следва строг ред, гарантиран от компилатора:

Базов клас (ако има наследяване): Първо се извиква конструкторът на родителя.

Вградени обекти (Членове): След това се конструират всички член-обекти.

Важно: Редът на конструиране зависи от реда на декларация в класа, а не от реда в списъка за инициализация (initializer list) на конструктора.

Тялото на конструктора на Контейнера: Най-накрая се изпълнява кодът в {} на самия клас.

3. Ред на деструкция Деструкцията е огледална на конструирането (Stack-like behavior - Last In, First Out):

Тялото на деструктора на Контейнера.

Вградените обекти (в обратен ред на декларацията).

Базовият клас.

ТЕМА 21: Заделяне на обекти от динамичната памет. Проблеми при взаимодействия.

1. Динамична памет (Dynamic Memory) Това е памет, която се заделя по време на изпълнение на програмата (Runtime), а не по време на компилация (като стека). Тя се управлява ръчно от програмиста.

В C (malloc/free):

malloc(size): Заделя суров блок памет с размер size байта. Връща void*, който трябва да се кастне. Не вика конструктори!

free(ptr): Освобождава паметта. Не вика деструктори!

Пример: int* ptr = (int*)malloc(sizeof(int));.

В C++ (new/delete):

new Type: Заделя памет И автоматично извиква конструктора на обекта.

delete ptr: Извиква деструктора И освобождава паметта.

Пример: int* ptr = new int;.

2. Работа с масиви

Заделяне: int* arr = new int[10]; – заделя масив и вика default конструктора за всеки елемент.

Освобождаване: delete[] arr; – Задължително се ползват квадратни скоби []! Ако ги пропуснете (delete arr), ще се извика деструктор само за първия елемент и поведението е неопределено (Undefined Behavior).

3. Проблеми при взаимодействие (Memory Management Issues) Когато обекти си взаимодействат и ползват динамична памет, възникват класически проблеми:

Memory Leak (Изтичане на памет): Забравяне на delete или загуба на указателя към паметта.

Double Free: Опит за изтриване на една и съща памет два пъти (често при лош Copy Constructor).

Dangling Pointer (Висящ указател): Указател, който сочи към вече освободена памет.

Решение: Използване на RAII (Resource Acquisition Is Initialization) и умни указатели (std::unique_ptr, std::shared_ptr), или правилно прилагане на "Rule of Three" (Деструктор, Copy-Ctor, Assignment Operator).

ТЕМА 22: Приятелски класове и функции. Статични членове.

1. Приятелски класове и функции (Friend) Това е механизъм в C++, който позволява контролирано нарушаване на енкапсулацията.

Дефиниция: Ако клас A обяви клас B или функция F за friend, те получават пълен достъп до всички негови членове (дори private и protected).

Синтаксис: friend class B; или friend int func(A obj); вътре в класа A.

Особености:

Приятелството не се наследява.

Приятелството не е транзитивно (Приятел на моя приятел не ми е приятел).

Приятелството не е симетрично (Ако А е приятел на Б, Б не е автоматично приятел на А).

2. Статични членове (Static Members)

Същност: Членове (променливи или функции), които принадлежат на класа, а не на конкретна инстанция (обект).

Независимост: Тяхната стойност и съществуване не зависят от това дали има създадени обекти (this не съществува в статични методи) .

Уникалност: Съществува само едно копие на статична променлива за всички обекти от този клас. Тя е споделена.

Приложение: Броячи на инстанции, споделени настройки, Singleton Pattern (като в примера с McDonald's app) .

TEMA 23: Припокриване на оператори (Operator Overloading).

1. Същност C++ позволява на потребителските типове (класове) да използват стандартните оператори (+, -, *, ==, [] и др.) по начин, естествен за тях.

Пример: Събиране на комплексни числа или конкатенация на стрингове (str1 + str2).

2. Синтаксис и правила Реализира се чрез функция със специално име operator@, където @ е символът на оператора.

Като член-функция: ObjectType operator+(const ObjectType& other);. Левият операнд е this.

Като външна функция: ObjectType operator+(const ObjectType& lhs, const ObjectType& rhs);. Често трябва да е friend.

3. Таблица на преобразуване | Израз | Като член-функция | Като външна функция | | :--- | :--- | :--- | | @a (унарно) | a.operator@() | operator@(a) | | a @ b (бинарно) | a.operator@(b) | operator@(a, b) | | a = b | a.operator=(b) | (Невъзможно - винаги е член) | | a[b] | a.operator[](b) | (Невъзможно - винаги е член) |

4. Ограничения

Не могат да се създават нови оператори (напр. ** за степенуване).

Не може да се променя приоритетът на операциите.

Не могат да се преопределят: . (точка), :: (scope resolution), sizeof, ?: (ternary).

TEMA 24: Преобразувания и операции-преобразувания.

1. Преобразуване на типове (Type Conversion) В C++ обектите могат да се превръщат от един тип в друг.

Имплицитно (Скрито): Автоматично, ако има подходящ конструктор (напр. A obj = 5; вика A(int)).

Експлицитно (Явно): Чрез cast оператори (static_cast, dynamic_cast).

2. Оператори за преобразуване (Conversion Operators) Класът може да дефинира как да се превърне в друг тип.

Синтаксис: operator Type() const { ... }.

Пример: Клас String може да има operator char*(), за да се ползва като C-стринг.

Тези функции нямат тип на връщан резултат (той се подразбира от името).

3. Двусвързан списък (Doubly Linked List) В контекста на структурите от данни (показано в слайдовете):

Това е структура, където всеки елемент (Node) пази данни и два указателя: Next (към следващия) и Prev (към предишния).

Позволява ефективно движение в двете посоки . TEMA 25: CASE: Обектно-структурирана програмна система - Плащане. UML, достъп и полиморфизъм. 1. Постановка на задачата (Сценарий) Целта е да се проектира система за обработка на плащания в онлайн магазин. Системата трябва да е гъвкава и да позволява плащане чрез различни методи (Кредитна карта, PayPal, Биткойн), без да се променя основната логика на магазина.

2. Проектиране на Класовата йерархия За да постигнем полиморфизъм, ни трябва Базов абстрактен клас.

Базов клас Payment: Той дефинира общия интерфейс – всеки метод на плащане трябва да има сума и функция "плати".

Производни класове (Concrete Classes): Реализират конкретната логика.

CreditCardPayment: Има номер на карта, CVV, валидност.

PayPalPayment: Има имейл и парола.

CashPayment: Най-прост, само сума.

3. UML Диаграма на класовете (Описание) UML (Unified Modeling Language) диаграмата визуализира структурата:

Payment (Abstract Class)

Полета: protected double amount;, protected string currency;

Методи: public Payment(double), public virtual void process() = 0; (чисто виртуална).

Наследяване (стрелка с празен триъгълник към Payment):

CreditCardPayment: Добавя поле private string cardNumber. Предефинира process().

PayPalPayment: Добавя поле private string email. Предефинира process().

4. Дефиниране на достъп, методи и пропъртита

Енкапсулация: Данните (като номер на карта) са private, за да не може външният свят да ги манипулира директно. Достъпът става чрез конструктора или сетери с валидация.

Права: Базовият клас предоставя protected достъп до сумата, за да могат наследниците да я виждат, но тя да остане скрита за main функцията.

5. Примерна имплементация (C++)

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// 1. Абстрактен базов клас
class Payment {
protected:
```

```
    double amount; // Достъпно за наследници
public:
    Payment(double amt) : amount(amt) {}
    virtual ~Payment() {} // Виртуален деструктор е задължителен!

// Чисто виртуална функция - задължава наследниците да я реализират
    virtual void processPayment() = 0;
};

// 2. Конкретен клас - Кредитна карта
class CreditCardPayment : public Payment {
private:
    string cardNumber;
public:
    CreditCardPayment(double amt, string cardNum)
        : Payment(amt), cardNumber(cardNum) {}

void processPayment() override {
        // Симулация на връзка с банка
        cout << "Processing Card payment of $" << amount
             << " using card ending in " << cardNumber.substr(cardNumber.length()-4) << endl;
    }
};

// 3. Конкретен клас - PayPal
class PayPalPayment : public Payment {
private:
    string email;
public:
    PayPalPayment(double amt, string mail)
        : Payment(amt), email(mail) {}

void processPayment() override {
        cout << "Processing PayPal payment of $" << amount
             << " for user " << email << endl;
    }
};
```

6. Полиморфизъм при изпълнението Това е сърцето на системата. Ние не работим с конкретни обекти, а с указатели към базовия клас.

Създаваме колекция (напр. vector).

Пълним я с различни обекти (new CreditCardPayment, new PayPalPayment).

Когато обхождаме списъка и викаме ->processPayment(), компилаторът използва v-table (таблицата с виртуални функции), за да извика правилния метод за съответния обект по време на работа (Runtime).

7. Преобразувания над типове (Type Casting)

Upcasting (Безопасно): От Дете към Родител. Payment* p = new CardPayment(...);. Става автоматично.

Downcasting (Рисковано): От Родител към Дете. Ако имаме указател Payment* p, но искаме да видим номера на картата (който е само в CreditCardPayment), трябва да преобразуваме типа.

В C++ се използва dynamic_cast. Той връща nullptr, ако обектът не е от търсения тип.

```
void printCardDetails(Payment* p) {
    // Опитваме да превърнем общото плащане в Картово плащане
    CreditCardPayment* card = dynamic_cast<CreditCardPayment*>(p);

    if (card) {
        // Успешно! Можем да ползваме методи на CreditCardPayment
        cout << "Card identified!" << endl;
    } else {
        cout << "This is not a card payment." << endl;
    }
}
```

ТЕМА 26: Разширяване функционалността на системата: въвеждане на интерфейси. Полиморфна употреба. 1. Проблемът с обикновеното наследяване Представете си, че искаме да добавим функционалност за Логване (записване на транзакцията във файл) или Сигурност (криптиране). Не всяко плащане се нуждае от едно и също. Ако сложим метод log() в базовия клас Payment, нарушаваме принципа за разделяне на отговорностите (SRP). Плащането не трябва да знае как да се записва на диска.

2. Решение: Въвеждане на Интерфейси В C++ интерфейсът е абстрактен клас, който съдържа само чисто виртуални методи и никакви данни.

Дефинираме интерфейс ILoggable.

Той съдържа метод virtual string getLogInfo() = 0;.

3. Наследяване на интерфейс (Множествено наследяване) C++ позволява един клас да наследи един "истински" родител и един или повече интерфейса.

Класът CryptoPayment може да наследи Payment (за да е платежно средство) И ILoggable (за да може да бъде записан в журнал).

Примерна имплементация (Разширение):

```
// Интерфейс
class ILoggable {
public:
    virtual string getLogString() = 0; // Чисто виртуална
    virtual ~ILoggable() {}
};

// Нов клас: Крипто плащане, което поддържа и плащане, и логване
class CryptoPayment : public Payment, public ILoggable {
private:
    string walletAddress;
public:
    CryptoPayment(double amt, string wallet)
        : Payment(amt), walletAddress(wallet) {}

// Реализация на Payment
    void processPayment() override {
        cout << "Crypto transfer initiated from " << walletAddress << endl;
    }
```

// Реализация на ILoggable string getLogString() override { return "LOG: Crypto transaction val=" + to_string(amount); } }; 4. Полиморфна употреба на интерфейси Силата на интерфейсите е, че позволяват на обектите да бъдат третирани според това какво могат да правят, а не какви са. Можем да напишем функция Logger, която приема ILoggable*. Тази функция може да логва:

Плащания (CryptoPayment).

Потребители (User клас, който също наследява ILoggable).

Грешки (Error клас).

Тези класове нямат нищо общо помежду си, освен че са подписали договора ILoggable.

```
// Функция, която работи с интерфейса
void writeToSystemLog(ILoggable* item) {
    cout << "Writing to disk: " << item->getLogString() << endl;
}
```

int main() { CryptoPayment* crypto = new CryptoPayment(500.0, "0xABC123"); // Полиморфизъм 1: Като плащане crypto->processPayment(); // Полиморфизъм 2: Като логваем обект writeToSystemLog(crypto); delete crypto; } 5. Обобщение на Разширяемостта (Open/Closed Principle) Този подход следва принцип от SOLID: Системата е отворена за разширение (можем да добавим BitcoinPayment или FaceIDPayment и да имплементираме интерфейси), но е затворена за промяна (не се налага да пренаписваме съществуващия код на модула Logger или PaymentProcessor). Това е висшият пилотаж в проектирането на обектно-структуриран код.