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

Класът представлява описание на тип, включващ едновременно данни и функции, които ги обработват. Данните се наричат **член-променливи**, а функциите – **член-функции**.
Дефиницията на един клас включва *декларация на класа* и *дефиниции на член-функциите*.
Докато класът представлява всички обекти от даден тип (пример кола), **обектът е инстанция на класа** (червена кола със счупен мигач).
Декларация на класа има следния синтаксис:

```
class <име-на-клас> {
        //Декларации на член променливи
        //Декларации на член функции
};
```

За да създадем обект от даден клас, използваме следната декларация:
triangle ob1 -> четем, обект ob1 от тип triangle.

Атрибут – описва състоянието на обектите;
Тип на данните – описва какъв е типът на информацията за даденият арибут; Behavior – описва какво може да прави обекта;
Капсулация (Encapsulation) - Ще се научим да скриваме ненужните детайли в нашите класове и да предоставяме прост и ясен интерфейс за работа с тях.
Наследяване (Inheritance) - Ще обясним как йерархиите от класове подобряват четимостта на кода и позволяват преизползване на функционалност.
Абстракция (Abstraction)-Ще се научим да виждаме един обект само от гледната точка, която ни интересува, и да игнорираме всички останали детайли.
Полиморфизъм (Polymorphism)-Ще обясним как да работим по еднакъв начин с различни обекти, които дефинират специфична имплементация на някакво абстрактно поведение.

2. Методи и параметри. Даннови членове и пропъртита.

Функциите, членове на един клас се наричат **методи** (**член-функции**).
Функциите са **блокове код**, които могат да се викат много пъти. Изпълняват се само когато са извикани и не заемат памет, когато са декларирани.
Функции, които не са дефинирани в библиотеките (разработчикът си ги пише) се наричат user_defined functions. В C++ функциите се състоят от две части:

- Декларация -> име на функцията, тип връщан резултат и параметри
- Дефиниця -> тялото на функцията

Методите могат да имат параметри, които подават допълнителна информация нужна за изпълнение.
Свойства (properties) наричаме характеристиките на даден клас. Класът дефинира какви полета да има обекта, докато обекта има свое множество стойности за съответните полета заделени в паметта.

3. Модификатори на достъп в клас.

Нивата на достъп са private, protected, public.
Private членовете са видими само в същия клас в който са дефинирани, докато **public** членовете са видими навсякъде.
Protected членовете са достъпни в класа, който са дефинирани и в класът наследник на този клас.
Ако базовият клас е деклариран като public в производния клас, всички public, private и protected компоненти на базовия клас се наследяват съответно като public, private и protected компоненти на производния клас.
По подразбиране, ако не се декларира модификатор за достъп, член-данните и член-функциите се считат за private.
Модификаторите на достъп се използват за контролиране на достъпа до член-данните и член-функциите на класа.

4. Accessor-методи. Mutator-метод.

Като за всяко поле се декларира модификатор за достъп, трябва и за всяко да се пишат get и set методи, наричани още accesor и mutator, с модификатор public.

Accessor (Getter) методите служат за прочитане на дадено поле или набор от данни в класа – read only.
Обикновено нямат параметри и връщат директно стойността на полето.
Имената на accessor методите трябва да започват с get.

Mutator (Setter) методите служат за променяне на стойността на едно или повече полета – write.
Не връщат стойност (тип void са) и имат параметър. Типът на параметъра трябва да съответства на типа на полето с което ще работим.
Имената им трябва да започват със set.

5. Методи на клас. Видове модификатори. Припокриване

Дефиниране на метод на класа в C++ изглежда по следния начин:

```
[<modifiers>] [<return_type>] <method_name>([param_list>)
{
    //тяло на метода
    [<return_statement>];
}
```

```
public int GetAge()
{
    return this.age;
}
```

С “return this.age” ние казваме от текущия обект (this) вземи (използването на точка) стойността на полето age и го върни като резултат от метода (чрез ключовата дума “return”).

При декларирането на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – public, protected, internal и private.
public - методите могат да бъдат достъпвани от други класове
internal - елемента на класа може да бъде достъпван от всеки клас в същото асембли (т.е. в същия проект във Visual Studio)
private (private се подразбира) - елементите не могат да бъдат достъпвани от никой друг клас, освен от класа, в който са декларирани.
Езикът C++ позволява функции с едно и също име да имат различно съдържание и действие.
Не могат да съществуват две или повече функции, които се различават само по тип на връщания резултат. Трябва или параметрите да са различни по брой, или да имат различни сигнатури. Достатъчно е един от параметрите да е с различен тип.

6. Програмни практики: цифров часовник: диаграми на класове и обекти; проект и програмни елементи.

Диаграми на класове и обекти са графични представяния на класовете и обектите в програмата, които показват как са свързани и какви данни и функции съдържат.
Те могат да бъдат създадени с помощта на специализирани програми като Visual Paradigm, Lucidchart или SmartDraw.
В UML (Unified Modeling Language) има няколко типа диаграми като: диаграма на компоненти, диаграма на дейности, диаграми на класове и обекти и други.
Диаграмите се използват за визуализиране на класовете.

UML диаграмите се делят на статични (отразяват класове, интерфейси...) и динамични (отразяват поведението на с-мата и взаимодействат м/у класовете и обектите по време на изпълнение).

7. Групиране на обекти. Колекции и итератори . Програмни практики: проект ‘notebook’; обектна структура, използване на колекции.

Колекциите са обекти, които сочат към група от обекти.
Те съдържат референции към данни от тип обект.
Всякакъв тип обект може да се съхранява в колекция.

Технологии за съхраняване на колекция:

- ***Масив (Array)*** - фиксиран размер, бързо и ефикасно за достъп, трудно за модифициране.
- ***Свързан списък (Linked List)*** - лесен за промяна, бавен за претърсване.
- ***Дърво (Tree)*** - лесен за промяна, съхранява елементите в ред.
- ***Хеш таблица (Hashtable)*** - използва се за индексиране на ключ който идентифицира елементите.

Видове колекции:
Collection - Контейнер от неподредени обекти. Повторенията са позволени.
List - Контейнер от подредени елементи. Повторенията са позволени.
Set - Неподредена колекция от обекти, в която повторенията не са позволени.
Map - Колекция от ключ/стойност по двойки. Ключът се използва за индекс на елемента. Повторение на ключове не се допуска.

8. Проектиране на класова йерархия в обектно-структурирана програма.

Принципът на замяната **“is-a”** ("е един от"): използва се за дефиниране на наследяване м/у класове.
Принцип “**is-like-a**” ("е като"): функционалност или поведение, което даден клас има и което е сходно с поведението на друг клас. Това може да бъде имплементирано чрез интерфейс или полиморфизъм.

Типове преобразувания :

- **Възходящо преобразуване на типовете (upcasting)** – преобразува се обект от наследен клас в обект на базов клас
- **Низходящо преобразуване на типовете(downcasting)** – обратно от upcasting
- **Преобразуване на типове с интерфейс (casting with interfaces)** – може да се извършва преобразуване м/у обекти, които имплементират интерфейс.

9. Оценка на качеството на кода. Свързаност и структурираност на кода.

За писане на качествен код е нужно да се спазват принципите за свързаност и структурираност (еднородност), за премахване на дублиращи се фрагменти и за практическо коментиране.
Добра практика също е да се визуализира програмата посредством UML (Unified Modeling Language).
Свързаността представлява връзка м/у отделните единици на една програма. В един добре структуриран код, тя трябва да се избягва възможно повече (чрез наследяване и абстракция).
Целта е да се постигне ниска свързаност, което позволява лесна поддръжка.
Структурираност означава целенасочеността на отделните компоненти на една програма.
Един компонент (клас или метод) трябва да изпълнява само една задача за която е предназначен.
В такъв случай се казва че има висока структурираност.
Ако един метод прави повече от една задача, то по добре е да създадем 2 метода които да изпълняват 2-те задачи по отделно.

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

Дублиращ се код в програма обозначава лош дизайн / структура на кода и прави поддръжката му трудна.

Когато е нужна промяна в кода, то тази промяна трябва да се направи на всяко копие на този код. А самият код става дълъг и неоптимизиран.

За оптимизиране може да се ползват изброен тип или като се разбие на по-малки методи, правейки кода по гъвкав, сбит и прегледен.

11. Проектиране на обектно-структуриран код.

Причината защо един код трябва да бъде обектно-структуриран и добре описан, е за да може всеки който го погледне лесно да разбере за какво става въпрос.

Главни проблеми при ООП програмирането са: структурираност, свързаност и дублирането на код.

Структурираност (еднородност) означава целенасоченост на отделните компоненти на една програма.

Един компонент (клас или метод) трябва да изпълнява само една задача за която е предназначен.

В такъв случай се казва че има висока структурираност.

Ако един метод прави повече от една задача, то е по-добре е да създадем 2 метода които да изпълняват 2-те задачи по отделно.

Свързаността представлява връзка м/у отделните единици на една програма. В един добре структуриран код, тя трябва да се избягва възможно повече (чрез наследяване и абстракция).

Добрият дизайн включва:

- разпределяне на отделните методи в подходящи класове
- всеки клас трябва сам да манипулира своите данни, и да се грижи за валидността им
- ниската свързаност води до локализация на промените.

Един метод е твърде дълъг когато изпълнява повече от една функция.
Един клас е твърде сложен когато изпълнява повече от една логическа цел.

12. Класове и обекти: разделяне на декларация и дефиниция. Създаване и унищожаване на обекти. Структура и обект.

Класът представлява описание на тип, включващ едновременно данни и функции (които ги обработват). Класовете могат да се наследяват. Данните се наричат **член-променливи** или **полета**, а функциите – **член-функции** или **методи**.

Дефиницията на един клас включва ***декларация*** *на класа* и дефиниции на член функциите.

Класът е шаблон, който се използва за създаване на обекти. Един клас се декларира посредством ключовата дума class. Синтаксисът на една декларация на клас е сходен с тази на една структура:

```
class <име-на-клас> {
    private:   //незадължителен
        //private променливи
    public:
        //public функции и променливи
} <списък от обекти>;
```

Функциите и променливите в даден клас са негови членове. По default те са private членове на класа и са достъпни само в неговите рамки.

Декларацията на клас е логическа абстракция, която дефинира нов тип.
Декларацията на обект създава физическа единица от такъв тип.
За създаване на обект се използва следната декларация:

```
int main() {
  mysclass ob;
  return 0;
};
```

Обектът, при създаването си може да приема различни по типове промеливи декларирани в конструктура на базовия клас. Броят им трябва да съотвества на броят, който е деклариран в конструктура.

Структура	***Клас***
Примитивен тип	Референтен тип
Има само полета	Поддържат наследяване
Не поддържа наследяване	Достъпа по подразбиране е private
Достъпа на полетата по подразбиране е public	

13. Конструктори и деструктори. Видове конструктори (виртуални и статични). Методи на клас.

Конструктурът на един клас се извиква всеки път, когато се създава обект от този клас.
Конструкторът има същото име като името на класа, към който принадлежи, и не притежава тип на връщан резултат.
Един клас може да има повече от един конструктор, с различни аргументи.
Ако в класът липсва дефиниран конструктор, то той се добавя автоматично от компилатора.
Общият вид на конструктора е:
<име-на-клас>::<име-на-клас>(<списък формални аргументи>)
{ //Тяло на конструктор }

Статичен конструктор - статичният конструктор на даден клас се изпълнява автоматично преди класът да започне реално да се използва. Извиква се най-много веднъж в рамките на програмата и не приема параметри и модификатори за достъп.

Виртуални конструктори - ако конструкторът е виртуален и се използва в собствения си клас, той се държи като статичен. С такъв конструкторът се постига по-голям полиморфизъм.

Деструкторът се извиква автоматично при разрушаването на обект от класа.
Деструктора служи главно за освобождаване на заделената за обекта динамична памет.
Деструкторът има същото име като класа, но предшествано от символа ~.
Деструкторът не може да връща резултат и не може да има аргументи.
В един клас може да има само един деструктор.
Ако липсва, то той бива генериран автоматично от компилатора.

~Point();

Функциите, членове на един клас се наричат **методи** (член-функции).
Дефиницията на член-функция има следния синтаксис:
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък с параметри>)
{ // Тяло на функцията }

14. Дефиниране на връзки. Взаимодействия на обекти по вертикала и хоризонтала.

Връзките между класовете показват какви са взаимоотношенията между 2 класа.
Има няколко типа връзки в ООП: “**is-a**” и “**has-a**”, “**uses-a**”, “**part of**”.

“part-of” – обекта и частта трябва да имат следната връзка:

- Частта може да принадлежи само на един обект.
- Частта има своето съществуване, управлявано от обекта.
- Частта не знае за съществуването на класа.
- Обектът е отговорен за съществуването на частите.

Пример: сърцето принадлежи на тялото.
Тяло – обект, сърцето – част

“has-a” - това е еднопосочна асоциация, която позволява двата обекта да се „срещнат“ за някаква работа и след това да се разделят.

Пример: адрес и човек

“is-a” – когато един клас наследява друг,като има възможност и да го разшири.

Пример: apple “is-a” fruit
Ябълката има характеристиките на овощия, като има и някои допълнителни.

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

Класът, който наследява свойства от друг клас се нарича подклас или производен клас.
Класовете дефинират тип, а подкласовете дефинират подтип.
Пример: имаме база от данни (масив) от указатели към обекти Item-и. Класът Item е общ за всичките разновидности обекти които може да има в базата данни.

```
Item *db[10];
db[0] = new Video(…);
db[1] = new CD(…);
db[2] = new DVD(…);
db[3] = new Video(…);
```

Тук всички новосъздадени обекти се падат под-тип на Item.

Предаване на параметри – определя методите, по които се предават данни между функциите.

- **Предаване на стойност (pass-by-value)** - актуалните параметри на функцията се копират във формалните параметри на функцита по време на обръщението.
- **Предаване чрез референция (pass-by-reference)** – за да предадем променлива по референция ние просто декларираме параметрите на функцията като reference (използваме символа &). Всички промени направени в референцията се предават на аргумента.

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

Наследяването в ООП ни позволява да моделираме една връзка м/у два обекта.
Обектът от който се наследява се нарича родителски клас, основен клас, базов клас или суперклас.
Обектът, който извършва наследяването се нарича дъщерен клас, производен клас или подклас.
Производният клас, наследявайки базовия придобива всичките му членове.
Когато един клас наследява друг се използва обшата форма:
<class име-производен-клас> : <тип-достъп> <име-базов-клас>
{ ... };
Тук тип-достъп е една от трите ключови думи: public, private или protected.
Ако спецификаторът е public, всички public членове на базовия клас стават public членове и на производния клас.
Ако спецификаторът е private, всички public членове на базовия клас стават private за производния клас. Ако отсъства спецификатор, той по подразбиране е private.

Полиорфизъм – способността на обекти, принадлежащи към различни класове да изпълняват метод извикан с еднакво име, всеки според подходящия начин.
Самата дума полиморфизъм означава много форми.
Главно се разделя на два вида в C++:

- Compile time полиморфизъм – този тип полиморфизъм се постига чрез overloading на функции или overloading на оператори.

- Runtime полиморфизъм – постига се чрез overriding на функция.

Полиморфизмът се използва при наследяване на класове.

17. Виртуализация и реализация на полиморфизма. Абстрактни класове и абстрактни методи. Виртуални функции и викане на виртуални функции на базов клас.

Виртуален метод се нарича предефиниран метод, който се извиква от наследен клас, когато се обръщаме за него като към базов клас.
По правило за да декларираме даден член като виртуален, трябва декларацията да е предшествана от ключовата дума virtual:

```
class myclass {
    …
 public:
    …
    virtual int area () { return (0); }
 };
```

Когато указател тип базов сочи към обект от наследен клас, ще се извика метода на наследения а не на базовия клас. Пример:
Animal *ptr = **new** Cat();
ptr->speak();
Ще се извика speak() методът на наследеният клас, а не на базовия както би станало без virtual.
Клас, който е деклариран или наследен от виртуална функция се нарича полиморфичен клас.

Абстрактни базови класове
Абстрактните класове са подобни на нормалните класове, с разликата че те съдържат методи които нямат дефинирано тяло в себе си. Това става с добавяне на “=0” в декларацията на виртуалния метод:

```
class myclass{
    …
 public:
    …
   virtual int area ()= 0;
};
```

Всички класове съдържащи най-малко една абстрактна функция са абстрактни класове.

Виртуалната функция представлява член-функция, която се дефинира в базовия клас и се предефинира в производния клас.
За да се създаде виртуална функция, преди декларацията на функцията трябва да се добави ключовата дума virtual.
Когато виртуална функция се предефинира в производен клас, ключовата дума virtual не е необходима.
Чрез виртуалните ф-ии се постига полиморфизъм по време на изпълнение.

```
int main()
{
   area *p;
   rectangle r;
   p=&r;
   cout<<”Rec has area: “<<p->getarea()<<”\n”;
   ...
   return 0;
}
```

18. Вграждане на обекти. Copy – конструктори. Присвоявания и обекти.

Вграденият обект е обект, който се създава отделно и след това се поставя в/у друг обект или програма.
Вградените обекти са самостоятелни и могат да работят независимо. Тези обекти са вградени за да работят в комбинация с други обекти или програми.
Предимства : възможност да бъдат лесно прехвърлени на различно място в родителския обект, докато връзките биха се счупили.
Вградените обекти могат да бъдат модифицирани без промяни на оригиналния изходен код.

Copy – конструктори

Това е конструктор, който създава обект, като го инициализира с данните од друг, вече създаден обект от същия тип.
Конструкторът за копиране има следния синтаксис:

```
classname (const classname &obj) {
        //body of constructor
}
```

Ако не дефинираме собствен copy конструктор, компилаторът автоматично генерира един по подразбиране.

Присвоявания и обекти
Присвояването на обекти включва прехвърлянето на стойности от един обект към друг.

19. Референтни парамети (const, non-const). Работа с референции. Връщане на референтни обръщения.

При използване на аргумент от тип reference, функцията получава адреса на фактическия аргумент. Този механизъм за предаване на параметри е известен като предаване чрез адрес (pass by reference) и води до следните два ефекта:
1. Промяната на аргумент в тялото на функция е промяна на фактическия параметър.
2. Без проблеми може да се предаде голям обект от даден клас, докато при предаване на аргументи по стойност, той се копира в стека при всяко извикване на функцията.
Ако използваме аргумент от тип reference, но не искаме да променяме стойността на фактическия параметър, можем да декларираме формалния параметър като const.
Употребата на аргументи от тип reference е подходяща при работа с класове. Тогава няма проблеми със съответствието на типове. Освен това размерът на обектите е значителен.
Механизмът не трябва да се използва, когато типът на фактическия параметър не може предварително да се определи.
Една функция може да връща резултат от тип reference или указател.

20. Конструиране на вградени обекти. Деструкция на вградени обекти.

Конструиране на вградени обекти:
1) Компилаторът обработва декларацията на обекта и заделя необходимата памет.
2) Всички вградени обекти са конструирани.
3) Извиква се конструктора на родителския клас.
4) Ob е конструиран вграден обект.
5) Функцията на конструктура на производния клас се извиква.

Деструкция на вградени обекти:

1) Извиква се деструктора на производния клас.
2) Ob е вграден обект и се разрушава.
3) Извиква се деструктора на родителския клас.
4) Вградените обекти се разрушават.
5) Паметта, заемана от родителския клас е освободена.

21. Заделяне на обекти от динамичната памет. Проблеми, породени от взаимодействията между обекти.

Всяка програма разполага с незаета памет, която може да използва при своето изпълнение.
Този резерв от памет се нарича динамична памет.
Тя не се свързва с име на променлива.
Тя е неинициализирана, следователно преди да я използваме трябва да я инициализираме.
Динамичната памет се заделя от операцията new, приложена към масив от обекти.
Например iny *pi = new int; заделя място за един обект от тип int, new връща указател към обекта и pi се инициализира с този указател.
Динамичната памет не е неограничена, тя може да се изчерпи.

Проблеми:

- Несъответствие в броя на заделянията и освобождаванията води до “изтичане” на памет (memory leak).
- Несъответствие в извикването на операторите за типове и масиви от типове, например извикване на **delete**, за памет, заделена с **new[]**.
- Опит за четене или писане на вече освободена памет или опит за повторно освобождаване на памет.
- Бавно заделяне (и освобождаване) на динамична памет;
- Неефективно използване на процесорите на машината поради неоптимизирани алгоритми за синхронизация

22. Приятелски класове и приятелски функции. Статични членове на клас.

Приятелските функции са функции, които имат достъп до частни (private) и защитени (protected) членове на клас, дори ако не са членове на същия клас. Функция или цял клас могат да бъдат обявени като приятели на друг клас.

Декларацията на приятелска функция се прави чрез ключовата дума friend в тялото на класа.
Приятелски функции се използват, когато е необходимо една функция да има достъп до private членовете на два или повече различни класа.
Приятелските функции се използват за повишаване на производителността.

Понякога всички обекти от един клас трябва да имат достъп до една и съща променлива. Във тези случаи е по-удобно да се използва една и съща променлива за всички обекти, отколкото всеки обект да поддържа свое копие на променливата.
Решение в такива случаи е обявяването на съответния член на класа за статичен. Статичният член действа като глобална променлива, но притежава следните две предимства:
1. Статичният член може да е скрит, докато глобалната променлива винаги е общодостъпна.
2. Статичният член не принадлежи на глобалната област на действие.
Член, който се използва за представяне на класа, става статичен, ако декларацията му започва с ключовата дума static.
Статичните членове:
- се създават, когато приложението, съдържащо класа се зареди;
- съществуват през целия живот на приложението;
- имат само едно копие, независимо колко обекта от този клас са създадени;
- Достъпват се през класа (НЕ МОГАТ ДА СЕ ДОСТЪПВАТ ПРЕЗ ИНСТАНЦИЯ)

```
class Orbiter {
    ...
  public:
    static int nCount;
};
```

Ако статичният член е деклариран като private трябва да се добави public статична променлива и да се работи с нея.
Статичните данни се използват в конструкторите, когато в run-time се решава какъв обект да се използва.
Всяка статична данна може да се инициализира само веднъж в програмата.

23. Предефиниране на оператори. Същност и ограничения. Предефиниране на аритметични операции.

Предефиниране на оператори в C++ позволява на програмистите да задават поведението на операторите за потребителски дефинирани типове данни. Това включва предефиниране на оператори като +, -, *, / и други за потребителски класове.

Същност:

- позволява на потребителите на езика да определят собствено поведение на операторите за техните класове.
- улеснява използването на потребителски типове данни в изрази, подобрява четимостта и удобството на кода.

Ограничения:

- не всички оператори могат да бъдат предефинирани за всички типове.
- не трябва да се злоупотребява с предефинирането, тъй като това може да доведе до неочаквано поведение на кода.

Унарни оператори

Унарните оператори се използват с един операнд. Това означава, че те действат върху една променлива, константа или израз.

Бинарни оператори

Бинарните оператори се използват с два операнда. Те са по-общо срещани и могат да се използват с числови стойности, символни низове, указатели и др.

Варианти на предефиниране на оператори:
- чрез дефиниране на член функция към класовата декларация;
- чрез глобална приятелска функция.

24.Преобразувания и операции.

Преобразованията е когато сменяме типът на един обект/променлива във друг (още се нарича cast-ване). Има 2 типа преобразования: автоматични и частни, за наще класове.

Автоматичните се получават при примитивните типове като int, float, double и т.н. Пример:

```
int a = 5.3; //a=5;
```

25. Софтуерни контракти. Пред и пост-условия. Инварианти. Примери.

1) Контракти

Контрактите осигуряват езиково-агностичен начин за изразяване на предположения в .NET програмите. Контрактите наподобяват пред и пост-условията и инвариантите.
Действивто им се изразява в проверка на документацията на външните и вътрешните API-та. Контрактите се използват за подобряване на тестването чрез проверка по време на изпълнение.

2) Инварианти

Това са условия или правила които не се променят в течение на времето (работата по проекта). Те се използват за да се гарантира съответствие на резултатите с определени изисквания и стандарти.
В случай на софтуерни контракти, инвариантите могат да включват: качеството на кода, поддържката и съвместимостта с други системи.

Пример:

```
struct Date {
  int day;
  int hour;

 __invariant() {
    assert(1 <= day && day <= 31);
    assert(0 <= hour && hour < 24);
  }
};
```

3) Пред и пост- условия

Пред-условията, които трябва да бъдат изпълнени преди изпълнението на самата програма.
Пост-условия – трябва да са true, за да се гарантира че, резултатът е правилен.
Те се използват за описание на това какво се очаква от функционалността и как тя трябва да влияв на други сегменти от програмата.

Пример:

```
 void write_sqrt(double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
```

26. Контракт и наследяване. Проблем с ограничаване на областта в дъщерен обект.

Софтуерните контракти могат да се наследяват във всички платформи, които ги поддържат, както и в .NET Framework.
Когато един клас наследява друг, дъщерният клас приема поведението, съдържанието и контрактите на родителя. Наследяването на контракти не създава проблеми/не обърква инвариантите и пост-условията. По-проблемно е за пред-условията.
Проблемът с ограничаването на областта във връзка с наследяването често се изразява чрез понятието "Liskov Substitution Principle" (Принцип на заместване на Лисков). Този принцип поставя изисквания за това как подкласът трябва да се възприема в програмата в сравнение с неговия суперклас.

27. Обектен дизайн : принципи на SOLID, open/closed принцип, принцип на регламентираната отговорност.

SOLID – популярен акроним, съдържащ инициалите на 5 принципа на софтуерния дизайн.

S (Single responsibility) - Всеки клас трябва да има една причина за съществуване.

O (Open/Closed) - Отворен за разширение, затворен за промени.

L (Liskov substitution principle) - Лисковият Принцип за заместване на единици от програма

I (Interface segregation) - Интерфейсите трябва да не бъдат дебели.

D (Dependency inversion) - Единици функционалност трябва да зависят от абстрактни единици, не една от друга. Абстрактни единици не трябва да

зависят от Детайли около една единица. Детайлите около единицата трябва да зависят от Абстракции.

28. Обектен дизайн: принцип на двойния dispatch (пренасочване) в run-time.

Double dispatch е техника която се използва в контекста на полиморфизма и се извиква за да „смекчи“ липсата от поддържката на мултиметод в ПЕ.
Тази техника позволява да се избере правилното действие за изпълнение, в зависимост от типа обекта.
Особено е полезен при имплементация на динамичен диспечер на събития.

29. Обектен дизайн : принцип на Лисков. Принцип на Лисков и контрактите в .NET.

Наследниците трябва да бъдат заместими от техните базови класове.
Правилна йерархия на класовете.
Методи или функции, които използват тип от базов клас, трябва да могат да работят и с обекти от наследниците без да се налага промяна.
Функциите, които използват указатели или референции към базовите класове трябва да могат да използват обекти от дъщерните класове, без да знаят това.
С други думи, ако извикваме метод, дефиниран в базовия клас на абстрактен клас, функцита трябва да се изпълни правилно в дъщерния клас. Когато използваме обект чрез интерфейс на неговия базов клас, дъщерният обект не може да се подчинява на пред-условия, които са „по-силни“ от тези в базовия клас.

30. Паралелизъм и асинхронност. Конструктори в C++. Асинхронни задачи, нишки и локални за тях данни.

Паралелизъм:
Паралелизмът се отнася до едновременното изпълнение на няколко задачи или подзадачи. Този процес може да бъде постигнат чрез използването на нишки (threads), процеси или други техники за едновременно изпълнение на код.

Асинхронност:
Асинхронността се отнася до изпълнението на задачи, без да се изчаква тяхното завършване преди да се премине към следващата задача. Това обикновено се постига с използването на асинхронни операции, обещания (promises), и асинхронни функции в програмирането.

Асинхронни задачи и нишки:
Стандартната библиотека на C++ предоставя поддръжка за асинхронни операции чрез std::async, std::future, и std::promise.
Нишки могат да бъдат създадени с std::thread и се използват за паралелно изпълнение на код.

Локални за нишките данни:
За предотвратяване на създаването на грешки при едновременен достъп до общи данни,
C++ предоставя инструменти като std::mutex, std::lock_guard и други за синхронизация на достъпа до ресурси. Всеки обект на std::thread има свои локални данни, които не се споделят с другите нишки.

31. Конструктори за синхронизация: атомични типове и условни променливи, мютекси и заключвания.

Конструктори за синхронизация са механизми, които се използват за контролиране на достъпа до обши ресурси на многонишкови програми.
Те се използват за защита на данните от конкуриращи потоци.
Също така могат да се използват за синхронизация на нишки и контролиране на поредността на изпълнение на операции.

Атомични типове – подобни на примитивните типове, но позволяват thread-safe модификация.
std::atomic е шаблонен клас, който позволява използването на атомични операции върху примитивни типове данни.
Гарантира, че операциите са изпълнени атомарно без нужда от използването на мютекси.

Условни променливи (std::condition_variable)
std::condition_variable се използва за чакане на определено условие да бъде изпълнено преди да се продължи изпълнението на кода.

Мютекси и заключения – елементи, които ни дават възможност да дефинираме thread-safe критични области.

std::mutex се използва за заключване на ресурс и предотвратяване на едновременен достъп от повече от една нишка.

32. Асинхронно програмиране: async и await. Обект awaitable. Резултат от асинхронна операция.

1) Async, Await.

Task-based Asynchronous Pattern (TAP) е нов модел за асинхронно програмиране в.NET Framework.
Асинхронното програмиране предотвратява забваяния в производителността и засилва цялостната отзивчивост на програмата.
Ключовите думи Async and Await в Visual Basic и async and await в C# са сърцето на async програмиране.
Използвайки тези 2 ключови думи, можем да използваме ресурси в .NET Framework или Windows Runtime, за да създаваме асинхронни методи почти толкова лесно както създаваме и синхронни методи.

```
public async Task DoSomethingAsync()
{ await Task.Delay(100); }
```

“async” активира “await” в този метод.Това е всичко което async прави!

Await е като унарен оператор – приема един аргумент (който е awaitable – асинхронна операция).

2) Awaitable

Има 2 типа awaitable:

- Task<T>
- Task One

Awaitable не е методът който връща тип.

3) Резултат от асинхронна операция

Можем да очакваме резултат от асинхронен метод който връща Task, не защото е асинрхонен а защото връща Task. Също така и от синхронен метод може да очакваме резултат.

33. Асинхронно програмиране: превключване на контексти, асинхронна композиция. Диаграма на изчислителния процес при асинхронни операции.

Превключването на контексти в асинхронното програмиране се отнася до смяната на изпълнението от една задача на друга, без да се изчаква първата да завърши.
Асинхронната композиция включва комбинирането на няколко асинхронни операции в по-голяма програма. Това позволява създаването на сложни асинхронни програми чрез комбиниране и композиране на по-малки, самостоятелни асинхронни компоненти.

34. Генетични (пораждащи) типове в обектното програмиране. Синтаксис. Начин на обработка в .NET среда. Разлика с шаблонизирани типове.

- Целта е сходна на целите на ООП – algorithm reusing.
- Механизмът е въведен в CLR на .NET.
- Реализациите да се отнасят за обекти от различен тип.
- Може да се създаде ‘генетичен референтен тип’ , ‘генетичен стойностен тип,
  ‘генетичен интерфейс’ и ‘генетичен делегат’. Разбира се и ‘генетичен метод’.

Въпреки подобността в функционалността, генеричните типове в .NET и шаблонизираните типове в C++ имат различен синтаксис и работят в различни среди.-Ясен код: рядко се налагат tape casts;
Подобрена производителност.

35. Генетични типове и наследяемост. Синтактично подменяне на генетичен тип. Обработка на генетични типове. Ограничители.

В контекста на генеричните типове в програмирането, наследяването се отнася до способността на генеричен тип да наследи или реализира интерфейси или други генерични типове.
Синтактичното подменяне на генеричен тип (type substitution) включва използването на конкретен тип вместо параметъра на генеричния тип. Това може да се постигне чрез инстанциране на генеричен клас или метод с конкретен тип.

Обработката на генерични типове включва създаване, манипулиране и използване на генерични структури (класове, методи и др.) в програмския код.
Ограничители в C# позволяват на програмистите да наложат определени ограничения върху типовете, които могат да бъдат използвани с генеричните класове или методи.

36. Lambda-expression. Елементи на ламбда-изразите. Функции-обекти.

Ламбда изразите представляват анонимни функции, които съдържат изрази или последователност от оператори.
Всички ламбда изрази използват ламбда оператора **=>**, който може да се чете като "отива в".
Идеята за ламбда изразите в C# е взаимствана от функционалните езици (например **Haskell**,**Lisp**, **Scheme**, **F#** и др.).
Лявата страна на ламбда оператора определя входните параметри на анонимната функция, а дясната страна представлява израз или последователност от оператори, която работи с входните параметри и евентуално връща някакъв резултат.
Обикновено ламбда изразите се използват като предикати или вместо делегати (променливи от тип функция), които се прилагат върху колекции, обработвайки елементите от колекцията по някакъв начин и/или връщайки определен резултат.

Пример:

```
int main()
{
   int m=0;
   int n=0;

   auto lambda = [&,n] (int a) mutable {
          m=++n+a;
   }

   lambda(5);
   std::cout<<m<<endl;
```

36A) Синтактични елементи в ламбда изразите

[capture list] (parameter list) -> return type {body of function}

‘Capture list’ (незадължително) – определя как и кои променливи от този блок от код се захващат от ламбда израза.

’Parameter list’ – определя входните параметри на ламбда израза, като регулярна ф-я.
‘Return type’ – определя типът на резултата от ламбда израза.
’Function body’ – съдържа кодът, който се изпълнява когато ламбда изразът бъде извикан.

Функции – обекти -> запазват състояние, но налагат синтактични усложнения свързани с дефинирането на нов клас.

37. Обекти в паметта – особености при разполагането и чести програмни грешки

В С обектите в паметта се разполагат с командите:
calloc();
malloc();
realloc().

Паметта се освобождава с функцията free()
В С++ за разполагане на обекти в паметта може да се използва и оператора new, а за освобождаване на паметта се използва оператора delete. В С++ може да се използват и командите от С.
malloc(size_t size);
*Локализира битовете и връща указател към локализираната памет.
* Паметта не е изчистена.
Free(void * p)
*освобождава паметта към която сочи p
*ако командата free(p) вече е била извикана може да се появи неопределено поведение.
* Ако p е NULL никаква операция не се извършва.

Чести грешки: Повечето С програмисти използват malloc() за локализиране на блокове с памет и предполагат, че паметта е нулирана.
Инициализирането на големи блокове с памет влиае на производителноста и не е винаги необходимо. По – добрия вариант за локализиране на памет е с memset() или извикването на calloc(), което нулира паметта.

38. Управление на памет в конзолен режим и в Linux системи: структури в паметта, повреждане на структурите при неправилно менажиране на памет.

В повечето Linux системи за алокиране на паметта се използва принципът на Doug Lea, който е доста бърз и ефикасен. Използва стратегия Best-Fit, т.е. преизползва
освободените (free) парчета (chunks) памет със най-малки загуби. При освобождаване
се сраства парчета в по-големи.
Техниката на Doug Lea се базира на двустранно свързани списъци от ОСВОБОДЕНИТЕ парчета.
При този метод, алокираните парчета имат следния формат: [size]{data}.
Когато освободиме парче памет, ние само го маркираме че то е освободено. На мястото на старта информация където е било{data} се записва друга служебна информация, а именно forward указател и back указател, както и още веднъж [size].

39. Техниката ‘frontlink’ за сриване на код. Пример.

При освобождаване на блок памет, той се слива в двойно свързан лист. Това се извършва от frontlink() macro(под Linux). Макрото съединява сегментите в намаляващ ред по големина.
В този случай недоброжелателят предоставя не адрес, а кратък код който цели да подлъже системата да изпълни функция предоставена от недоброжелателя вместо нейна.

Уязвим код:

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char*argc[]){
char *first, *second, *third;
char *fourth, *fifth, *sixth;
first = malloc(strlen(argv[2]) + 1)
second = malloc(1500);
third = malloc (12);
fourth = malloc (666);
fifth = malloc(1508);
sixth = malloc(12);
strcpy(first, argv[2]);// получава се препълване на буфера
free(fifth);// във forward pointera на петия блок се слага адресс към фалшив блок.
strcpy(fourth, argv[1]);
free(second);
return(0);
}
```

Във фалшивия блок е съхранен адрес към пойнтер на функция. Този поинтер може да сочи към първата извикана деструктор функция(нейния адрес може да бъде намерен в сектора dtors на програмата).

40 и 41. Препълване на буфер. Поглед отвътре. Техники на вмъкване на код. Пример.

Препълване на буфер е явление, което настъпва когато процесът пише върху буфер извън заделената за тази програма памет. Така се пренаписва неправомерно съседна памет, където може да се съдържат други променливи, или функции, което може да доведе до грешки при достъп на тази памет, неправилна работа на програмата или пробив в сигурността.
Най-често се причинява при липса на проверка за размерността/граници на масива, особено при въвеждане на данни.

Пример за грешки при Doug Lea подредба на паметта:

```
#define BADSTR "……"
int main() {
   char *first, *second;
   first = malloc(666);
   second = malloc(12);

   free(second);
   strcpy(first, BADSTR); //BADSTR съдържа вреден низ
   return 0;
}
```

Когато заделим first който е по голямо парче 666 байта, очакваме непосредствено след това да бъде заделено second тъй като е по-малко парче.
След което с free (second) това парче бива освободено и вкарано в свързаният списък със свободни парчета, а старата data от second бива пренаписана със служебна информация forward pointer и back pointer.
След което се изпълнява strcpy(first, BADSTR), която цели да копира втория аргумент, в полето first без никакви проверки.
Ако подадем BADSTR който е по-голям от заделената памет за first, примерно BADSTR е 668 байта, а first е 666, то ще бъдат пренаписани 668 байта от началото на first надясно, т.е. 2 байта ще бъдат пренаписани в/у следващата клетка.
По-подобни начини може да бъде изменена/повредена структората на паметта, да бъдат пренасочени собствени функции към функциии изпълняващи зловреден код или най-малкото да променим действието на

програмата. Такива атаки може да използват **unlink**, **frontlink**, двойно освобождаване на паметта и т.н. Дефрагментацията и слепването също спомагат за това.
Повторно освобождаване на вече освободена памет води до объркване на логиката на списъците, което също е опасно.

42. Динамично управление на памет в Windows.

HeapAlloc() – функцията се използва за взимане на памет от dynamic heap. Heap-а е структура от данни, която се използва за динамично управление на паметта, както за взимане така и за освобождаване на памет с произволна големина.
Има следния синтаксис:

```
LPVOID HeapAlloc (
        HANDLE hHeap,
        DWORD dwFlags,
        SIZE_T dwBytes
);
```

HANDLE hHeap – номер на Heap-а от който се взима паметта
DWORD dwFlags – допълнителни опции за взимането на паметта
SIZE_T dwBytes – големината на паметта която трябва да се вземе в байтове

VirtualAlloc() – за взимане на памет от виртуалната адресна област.
Позволява заделяне или преоразмеряване на виртуална памет и контролиране на достъпа до нея.

43. Служебни структури в динамичния менижмънт на паметта в ОС Windows.

В операционните системи от семейството на Windows, динамичният мениджмънт на паметта е реализиран чрез функции и структури, предоставящи операции за заделяне и освобождаване на памет. Една от важните структури в този контекст е служебната структура за управление на блоковете от памет (Memory Block Header). Когато се използва операторът new или функциите като malloc() в C/C++, операционната система алокира блок от памет, който съдържа не само данните, но и служебна информация. Тази информация се съхранява преди или след фактическия блок и обикновено включва: Дължина на блока: Размерът на заделената памет (без служебната информация). Статус на блока: Показва

дали блокът е свободен или зает. Указатели към следващия и предишни блок в паметта: Ако блокът е част от списък от свободни блокове, тези указатели показват към следващия и предишни свободни блокове.

44. Препълване на буфер в Windows и атаки, базирани на това. Пример. Техники за вмъкване на код и пренасочване на управление

Buffer overflow се случва, когато в буфер или друго място за съхранение на данни се сложи повече отколкото буферът може да съхранява.
В зависимост от операционната система и от това какво точно представлява допълнителната информация, с която се препълва буфера, това може да се използва от злонамерени хора, за да причинят сривове в системата, или дори да изпълнят произволен код. В миналото голяма част от пробивите в сигурността са възникнали именно благодарение на такива проблеми.
Един от първите значителни пробиви в сигурността, свързан с buffer overflow се е случил през ноември 1988 г и се е наричал „червеят Morris". Той е използвал програмата finger на Unix-базираните компютърни системи.
Буфер – това е непрекъсната част от паметта, например масив или указател в C. Поради липсата на автоматично определяне на границите, може да се пише и отвъд границата на буфера.

```
int main() {
int buffer[100];
buffer[135] = 10;
 ...
 }
```

Горната програма е валидна и всеки компилатор би я компилирал без да даде съобщение за грешка. Обаче програмата се опитва да пише отвъд заделената за буфера памет, което може да доведе до неочаквано поведение от нейна страна.
За да се опише защо това може да създава проблеми, трябва да се спомене какво представлява един процес в паметта.
Процес – това е програма в хода не нейното изпълнение. Една изпълнима програма, която се намира на хард диска съдържа: набор от инструкции, които трябва да се изпълнят от процесора (code segment); данни достъпни само за четене (data segment); глобални и статични данни, които могат да се достъпват от програмата в хода на нейното изпълнение; brk pointer, които следи за заделената памет; локални променливи на функции (автоматични

променливи, които се създават в стека, когато функцията се изпълнява и се изтриват автоматично, когато функцията приключи).
Стек – непрекъснат блок от паметта, които съдържа данни. Указател към стека (stack pointer – SP), сочи към върха на стека. Дъното на стека се намира на фиксиран адрес в паметта. Размерът му се променя динамично от ядрото. **Процесорът** имплементира инструкции са добавяне на елемент в стека (push) и изваждане на елемент от стека (pop). Стекът се състои от логически единици наречени слоеве. Когато се извика функция, параметрите на функцията, се слагат в стека от дясно на ляво. След това в стека се слага адреса на връщане (return address, или адресът който трябва да се изпълни след като функцията се върне) и frame pointer-a (той указва локалните променливи, защото те са на константно разстояние от него). Локалните автоматични променливи се слагат след frame pointer-a.

```
void f (int a, int b, int c) {
 char buffer1[5];
 char buffer2[10];
}
int main() {
 f(1,2,3);
}
```

Стекът на функцията f, изглежда така:
Както се вижда за buffer1 са заделени 8 байта, а за buffer2 – 12 байта, тъй като адресируемата памет е равна на думата (тоест 4 байта). FP се използва за да бъдат достъпни buffer1, buffer2, a, b, и c, като всички тези променливи се изчистват от стека, след като програмата приключи.

```
void function(char *str) {
 char buffer[16];
 strcpy(buffer,str);
}
void main() {
 char large_string[256];
 int i;
 for( i = 0; i < 255; i++)
 large_string[i] = 'A';
 function(large_string);
}
```

Тази програма е класически пример за препълване на буфера. Тя ще има неочаквано поведение, защото низ с дължина 255 байта се копира в низ с дължина 16 и по този начин буфера се препълва и се пренаписват данните

от FP и данните от return address-а (тоест адреса, където трябва да се отиде след изпълнение на функцията), че даже и данните за параметрите на функцията. По този начин се „поврежда” стека на програмата и тя най-вероятно няма да може да продължи нормалното си изпълнение,
като в най-лошия случай може да се стигне и до изпълнение на произволен код, или извикване на произволна програма. В конкретния случай стека на функцията се препълва с А-та, шестнайсетичната стойност на която е 0x41, което означава че return address-а вече е 0x41414141, което дава segmentation violation, защото това се намира извън адресното пространство на процеса. По този начин чрез препълване на буфера може да се променя return address-а на дадена функция.
Основни концепции за защита от препълване на буфера:
- писане на сигурен код – това е единственият начин да се елиминира напълно проблема с препълването на буфера.
Библиотечните функции на C : strcpy (), strcat (), sprintf () и vsprintf (), работят с null-terminated низове и не проверяват за излизане отвъд границите. Същия проблем имат и scanf(), и gets().
Затова най-лесният начин за защита от buffer overflows е да не им се позволява да възникват.
Да не се позволява да си изпълняват инструкции от стека, тъй като „вредният” код се намира именно там, а не в сегмента за код (code segment). Този метод е доста труден за инплементация.
Самите компилатори могат да познават и да предупреждават за използване на опасни функции.
Използване на програми като Stack Guard, които поставят някаква специална дума в до return address-а в стека и ако тази дума е променена, това означава, че и return address-а е бил променен и Stack Guard не позволява изпълнението на командата (такава програма, или поне нещо подобно има вградена в Windows 2003 Server).
Динамични проверки по време на изпълнението на програмата.
При този метод една програма има ограничен достъп, с цел да се предотвратят евентуални атаки.

45. Съпоставяне на файл с Оперативна Памет. Програмни практики при управление на паметта.

Memory mapped file

(асоцииране на файл с адресно пространство от паметта)

След това, когато се заяви достъп до страница от паметта, memory manager я чете от диска и пъха в RAM. Ето как се развива процесът:

```
HANDLE hFile = ::CreateFile(....)        //създаваме file handle
HANDLE hMap = ::CreateFileMapping(hFile, ...); //манипулатор на  file mapping object
LPVOID lpvFile = :: MapViewOfFile(hMap,.. ); // "map" на целия или на част от  файла
DWORD dwFilesize = ::GetFileSize(hFile,..)
// използваме файла
...
::UnmapViewOfFile(lpvFile);
::CloseHandle(hMap);
::CloseHandle(hFile);
```

Два процеса могат да ползват общ hMap, т.е. те имат обща памет (само за четене). lpvFile разбира се е различен.

За да имаме обща памет:

(функцията *GlobalAlloc(..., GMEM_SHARED,..);* в Win32 не прави shared блок, както беше в Win16.)

Обща памет, но не от общ файл:
както по-горе, без CreateFile() и с подаване на парам 0xFFFFFFFF вместо hFile. Създава се поделен file-mapping обект (напр м/ду процеси) с указан размер в paging файла, а не като отделен файл. (MFC няма поддръжка на този механизъм – CSharedFile прави обмен на общи данни през clipboard.)

* Няма разлика м/ду глобален и локален heap. Всичко е в рамките на 2GB памет за приложението.
* ползвайте ф-иите за работа с памет на C/C++ и класовете, ако нямате специални изисквания;
* създавайте свои, или викайте API ф-ии при по-специални случаи;

---

* има 2 вида heap: 1 авт. заделен от ОС за приложението(GlobalAlloc(),която вика HeapAlloc(),или по-лесно- работа с malloc/free,или още по-лесно- new/delete) и 2. собствени heap блокове:
  - =създаване hHeap = HeapCreate(,,.размер);
  - //може синхронизиран достъп до хипа от повече от 1 thread в рамките на процес
  - = заделяне памет от създаден pHeap = HeapAlloc(hHeap, опции, размер);
  - = освобождаване HeapFree();

Някои съвети при работа със собствен heap

* Създавайте локален heap в рамките на своите класове (по 1 за клас)
** избягва се се фрагменацията при продължителна работа.
** нараства безопасността, поради изолацията в рамки на процес
**позволява модифициране на new, delete операторите, конкретно за клас, в рамкте на конструктора. Съблюдавайте схемата:
1. ако не е създаден, създава се private heap и се инициализира свързан с него брояч (на използванията)
2. заделят се необходимия брой байтове;
3. инкрементира се брояча.

По аналогична схема се предефинира и операция delete