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*.
Обект – съдържа данни и инструкции;
Атрибут – описва състоянието на обектите;
Тип на данните – описва какъв е типът на информацията за даденият арибут; Behavior – описва какво може да прави обекта;
Метод – съвкупност от инструкции;
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**).
Обектите имат операции, които могат да бъдат извиквани - **методи**. Методите могат да имат параметри, които подават допълнителна информация нужна за изпълнение**.**
Свойства (properties) – така наричаме характеристиките на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект или да са споделени между всички обекти от тип даден клас.

3. Модификатори на достъп в клас.

Нивата на достъп са private, protected, public. При private членовете са видими само в същия класа, а при public са видими за всички. Protected членовете са достъпни в калса, 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.
Ако е необходимо да се предаде аргумент на конструктора на базовия клас, то всички необходими аргументи за базовия и за производния клас се предават на конструктора на производния клас.

4. Accessor-методи. Mutator-метод.

По подразбиране данните на обекта са достъпни за останалите обекти, но това нарушава неговата самостоятелност (object's privacy). Не бива да се променя състоянието на обекта без да се познава. По-добрият начин е да се включат методите "set" и "get" в дефиницията на класа. Те се наричат Mutator и Accessor методи.
•**Mutator** метода модифицира състоянието на обекта (променя стойностите на атрибутите му)
Например: labelName.setForeground(Color.white)
Mutator (set) методи често имат 1 или повече аргументи и служат за променяне на стойността на 1 или повече полета – write. Обикновенно тези методи не връщат стойност (са void тип). Типът на аргументите трябва да съответства с типовете на полетата с които ще работиме. По този начин макар че полето е private, може да променим неговата стойност при нужда. Използването на Mutator методи ни позволява да сложим проверки за валидност на подаваните данните при променяне на полетата.

```
void setPrice(float pr) {
  if (pr>0.0) price=pr;
};
```

•**Accessor** метода запитва обекта за някои негови данни без да го променя
Например: labelName.getFont()
Accessor (get) методите често нямат параметри и служат за прочитане на дадено поле или набор от данни в класа – read only. Обикновенно връщат директно стойността на полето. За тази цел типът на стойността

която връща функцията трябва да съответства на типът полето. Така макар че полето е private, може да се вземе неговата стойност отвън класа, а при липсата на такъв метод, може да бъде скрита.

```
float getPrice() {
  return price;
};
```

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

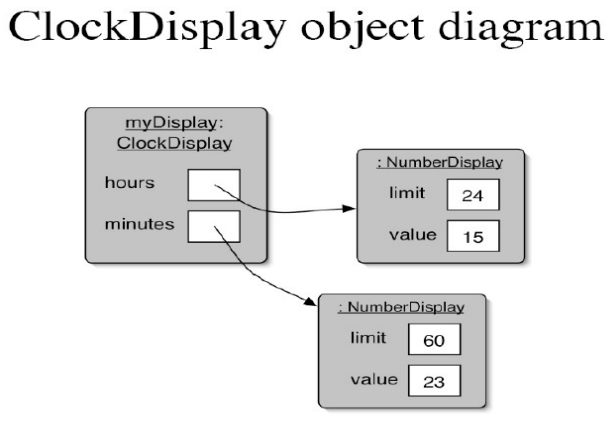
```
// Method definition
[<modifiers>] [<return_type>] <method_name>([<parameters_list>])
{
      // ... Method's body ...
      [<return_statement>];
}
```

Работата, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "{" и "}".
Ключовата дума **this** е референция към текущия обект, към който се извиква метода.

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

С "return this.age", ние казваме "от текущия обект (this) вземи (използването на оператора точка) стойността на полето age и го върни като резултат от метода (чрез ключовата дума return)" (този достъп е възможен само от нестатичен код).
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, независимо дали другите класове са декларирани в същото пространство от имена, в същото асембли или извън него;
internal - елемента на класа може да бъде достъпван от всеки клас в същото асембли (т.е. в същия проект във Visual Studio), но не и от класовете извън него (т.е. от друг 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. При обръщение към функция с дадено име се зарежда тази от едноименните функции, която отговаря най-точно по брой, тип и ред на аргументите. Не могат да съществуват две функции или повече, които се различават само по тип на връщания резултат. Трябва или параметрите да са различни по брой, или по подредба или типовете в дадената последователност (т.е. 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.



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. Колекциите съдържат референции към данни от тип обект. Всякакъв тип обект може да се съхранява в колекция.
Collections Framework – интерфейси;имплементация;алгоритми
Технология на съхраняване на колекцията:
Масив (Array)-Фиксиран размер, бързо и ефикасно за достъп, трудно за модифициране.
Свързан списък (Linked List) - Елементите имат референция за упътване към следващия елемент, 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. Елементите са извличат от хеш таблицата чрез използване на ключ на елемента.

Видове колекции:

Collection-Прост 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.

Using the collection

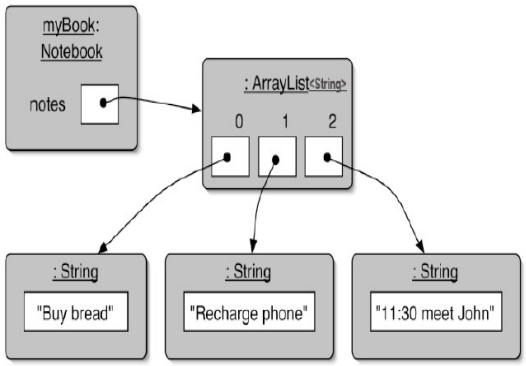
```
public class Notebook
{
    private ArrayList<String> notes;
    ...

    public void storeNote(String note)
    {
        notes.add(note);
    }

    public int numberOfNotes()
    {
        return notes.size();
    }

    ...
}
```

Index numbering



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

Принципът на замяната **“is-a”** ("е един от", "е"): ако може да се управлява само предефиниране на методите, тогава базовия и производния клас имат общ интерфейс, и техните обекти се използват като взаимозаменяеми (например, USB -устройства с флаш памет от различни производители).

Принцип “**is-like-a**” ("е като", "е и"):ако новият клас наследява множество от методив своя интерфейс,обектите на новия и стария клас са взаимо заменяеми "в еднапосока" (например, цифров фотоапарат се свързва чрез интерфейс USB "е и" ("е като") флашпаметта, но също така може да прави снимки.

Класова диаграма:

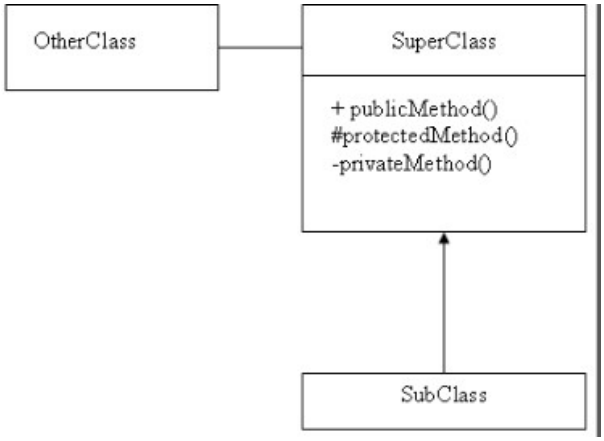
–Обединяване на данни и функционалност в един клас, позволява да се опише и моделира "концепцията" на предметната област;

–Създава възможност за описание на подобни концепции с уточняване само на различия от началната (базова) концепция;

-Началната концепция се представя от базовия клас (суперкласа, родителския клас);

–Промени, се указват в наследени класове (производни, дъщерни [под] класове);

–Базовият тип съдържа всички характеристики и действия, общи за всички типоове, наследени от него.


OtherClass
SuperClass
+ publicMethod()
#protectedMethod()
-privateMethod()
SubClass

Типови преобразувания- оперират с обекта като обект на другия тип;
Възходящо преобразуване на типовете (upcasting)-оперира с обект на производен тип като с обект на базовия тип (движещ се по диаграмата на наследяване);
Низходящо преобразуване на типовете(downcasting)-оперира с обекта, като представител на конкретния тип.

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

Когато се пише една програма трябва да винаги да пишем код който лесно да може лесно да се поддържа и променя. За тази цел, освен прегледността и подредбата на кода, трябва да се съобразяваме с понятията свързаност (coupling) и структорираност (cohesion) за да имаме качествен код.
Свързаност имаме когато имаме връзки м/у отделни компоненти на една програма. Когато 2 класа имат много връзки един с друг (използват взаимно методите си), то те са силно свързани. Това води до трудно проследяване или промяна на кода без да се повлияе на други класове. При такива случаи може да се проектира по добре като слеем 2-та класа в един. Цели се ниска свързаност, което позволява лесна поддръжка.
Структорираност 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 структорираност. Ако един метод прави повече от една задача, то по добре е да създадем 2 метода които да изпълняват 2-те задачи по отделно. Това спомага за универсалността на компонента, както и лесното й прочитане и поддържане.

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 във 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.
Пример:
//keys е масив който съдържа всички клавиши дали са натиснати или не.

```
void KeysPressed(bool keys[]) {
  cout << “Pressed keys are: \n“;
  if (keys[VK_ENTER]) cout << “Pressed Enter \n”;
  if (keys[VK_LEFT]) cout << “Pressed Left \n”;
  if (keys[VK_UP]) cout << “Pressed UP \n”;
  … //Още клавиши
  if (keys['A']) cout << “Pressed A \n”;
  if (keys['B']) cout << “Pressed B \n”;
  … //Още букви
}
```

Ако в този пример се опитаме да направим превод на български, трябваше на всеки ред да променим “Pressed “ със “Натиснат ”.

Пример- оправен:

```
void KeysPressed(bool keys[]) {
  cout << “Pressed keys are: “;
  for(int i; i<= KEYSMAX; i++)
     if (keys[i])  //Ако i-тият клавиш е натиснат
        cout << „Pressed “ << getKeyName(i) << „ \n”;
}
```

За да се избегне повтарянето на цели методи в сходни класове се използва наследяването на класове, където от даден базов клас се наследяват всичките функции и полета, и няма нужда да се имплементират наново в новият ни клас, което ни позволява да опишеме само нещата по които се различава новия клас от базовия.

11. Проектиране на обектно-структуриран код.

Главни проблеми при ООП програмирането са: структорираност (cohesion), свързаност (coupling) и дублирането на код.

При структорираността, проблемът е всеки метод да изпълнява само една функция, за което е предназначена. Когато става въпрос за класове, те трябва да бъдат ясно разграничими и дефинирани единици, удводлетворяващ нуждата от него.
При свързаността, имаме класове които се използват взаимно. Ако има прекалено много връзки между 2 класа, това говори за лош дизайн на кодът.
При дублирането на код се усложнява поддръжката на кода и говори за лош дизайн.
Добрият дизайн включва:
Разпределяне на отделните методи в подходящи класове.
Всеки клас трябва сам да манипулира свойте данни, както и да се грижи за валидността им.
Ниската свързаност води до локализация на промените- ако е нужна промяна трябва да засегне колкото се може по малко класове. Ако промяната е трудна – лош дизайн.
Good code: no duplication, high cohesion, low coupling.
Един метод е твърде дълъг когато изпълнява повече от една функция. Един клас е твърде сложен когато изпълнява повече от 1 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 полетата и методите.
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>;
```

В декларацията на класа обектите могат да се декларират и по-късно.Функциите и променливите в даден клас са негови членове. По default те са private членове на класа и са достъпни само в неговите рамки. За да се декларират публични членове е необходимо да се използва ключовата дума „public:”.По принцип за дефинирането на член-функция се използва следната форма:

```
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък –параметри>)
{
  //тяло на функцията
}
```

Декларацията на клас е логическа абстракция, която дефинира нов тип. Тя определя как ще изглежда един обект от този тип. Декларацията на обект създава физическа единица от такъв тип. Тоест, един обект полетата му заемат памет и методите му могат да се извикат, докато на класът-не. Всеки обект от даден клас притежава свое собствено копие от всяка променлива, декларирана в този клас.
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 съответства на броят, който е деклариран в конструктура.
Обектът освобождава паметта когато излезе извън обсега на блока или когато бъде ръчно освободен чрез delete (ако е бил заделен с new), при което се извиква неговият деструктор.
Един обект може да бъде присвоен на друг обект, като полетата на единия се копират в полетата на другия. Трябва да се внимава в случаите когато има полета-указатели и освобождаването на обектите които сочат.

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

Конструкторът на един клас се извиква всеки път, когато се създава обект от този клас. И поради това всякаква инициализация, която трябва да се извърши за даден обект, може да се изпълни автоматично от функцията конструктор.

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, като той само заделя памет за член-променливите без никаква инициализация, и без параметри. 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; гарантирано е извикан преди да се създаде първата инстанция на класа; гарантирано е извикан преди първия достъп до статичен член на класа (поле или метод). Използва се за инициализация на статичните член-променливи на класа. Не приема параметри и модификатори на достъпа.
Виртуални конструктори - ако конструкторът е виртуален и се използва в собствения си клас, той се държи като статичен. Такъв конструкторът може да се вика и като референция към клас. С виртуални конструкторът се постига по-голям полиморфизъм.
Деструкторът се извиква автоматично при разрушаването на обект от класа. Деструктора служи главно за освобождава 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, като в него просто се освобождават член-променливите на класа. Общият вид на деструктора е:
<име-на-клас>:: ~<име-на-клас>()
{ //Тяло на деструктор }
Обявяването на **деструктора** на един клас за **виртуален** води до това, че всички деструктори на класове по-надолу от него в йерархията се разглеждат като виртуални и всичките се извикват в правилния ред. Конструкторите и деструкторите не се наследяват автоматично и трябва ръчно да се предефинират при наследени класове.
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 2 типа връзки в ООП: “is-a” и “has-a”.
“is-a” е когато един клас наследява друг клас. Тоест наследеният клас има същата функционалност и структора като базовият клас, като е възможно да го разшири.
Пример: apple наследява fruit. Тогава може да кажем: apple IS A fruit.
“has-a” е когато в един клас се съдържат членове на друг клас. Когат се ползва такава връзка, обикновенно трябва ръчно да заделим този допълнителен обект в паметта, след което не трябва да се забравя да се изчисти паметта от него. Честа грешка е да се изтрива главния обект, без заделените от него съдържани обекти -> memory leak.

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

Класовете дефинират тип, а подкласовете дефинират подтип. Базовият клас се явява главен и той може да бъде оприличен на всеки един от наследяващите го класове, т.е. е логически свързан със него и практически наследеният клас може да го замести в даден момент.
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 стаойност (pass-by-value)- актуалните параметри на функцията се копират във формалните параметри на функцита по време на обръщението;

- Предаване чрез референция (pass-by-reference) – формалните параметри на функцията са алтернативни имена на съответните фактически параметри по време на обръщението към функцията.

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

Основна характеристика на C++ е **наследяването**. То позволява да се създават класове, получени от други класове, така че те автомачино включват някои от свойте „родителски" членове, както и свойте собствени.Характерно за **наследяемостта** е, че полетата и методите (без private) се копрат и в наследяващия клас, което предотвратява повторното писане на телата им в наследяващите класове.
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.
Спецификаторът за достъп определя по какъв начин елементите на базовия клас се наследяват от производния клас. Когато 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. И в двата случая всички private членове на базовия клас си остават private за него и не са достъпни от производния клас.
Спецификаторът за достъп protected е еквивалентен на спецификатора private с едно единствено изключение – protected членовете на базовия клас са достъпни за членове на всеки на всеки клас, който е производен на базовия клас. Извън базовия клас или неговите производни, protected членовете са недостъпни. Когатоедин protected член на клас се наследи като public от производен клас, той става protected член на производния клас. Ако базовия клас се наследи като private, един protected член на базовия клас става private член на производния клас. Един базов клас може да бъде наследен и като protected от един производен клас. В такъв случай public и protected членовете на базовия клас стават protected членове на производния клас.
Ако отсъства спецификатор, то по подразбиране той е private за клас и public за структора.
Тъй като конструкторът не се наследява автоматично, трябва да го предефинираме наново или да го наследим от базовия клас. Синтаксисът за наследяване е:
<производен-конструктор>(<списък-аргументи>): <базов-клас>(<списък-аргументи>)
{ …}
Същото се отнася и за деструкторите.
Съществуват два начина, по които един производен клас може да наследи повече от един базов клас. Първо, един производен клас може да бъде използван като базов за друг производен клас, като по този начин се създава многостепенна класова йерархия. В този случай за оригиналния базов клас се казва, че е индиректен базов клас на втория производен клас. Второ, един производен клас може директно да наследява повече от един базов клас. В този случай два или повече базови класа се използват за създаването на производен клас.
Когато един производен клас директно наследява множество базови класове, се изреждат със запетайки така:
class <име-производен-клас>: <тип-достъп> <база1>, <тип-достъп> <база2>, ... N
{ ... };
В случая когато производния клас наследява директно множество базови класове е възможно да възникне проблем, а именно да се наследи два или повече пъти даден базов клас, намиращ се по-нагоре в йерархията. Проблемът може да се реши чрез наследяване на базовия клас като виртуален. Това предотвратява присъствието на две или повече копия на базовия клас във всеки следващ индиректен производен клас. Когато се създава подобен производен клас, ключовата дума virtual се поставя пред спецификатора за достъп до базовия 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. При private е по-разично. Когато клас наследява друг клас като private то той наследява всички public членове като private.
Не се наследяват конструктори, деструктори и приятелски функции. Тъй като конст. не се наследяват автоматично и трябва да се предефинират наново, като аргументите на конструкторът от наследеният клас трябва съвпадат или разсширяват аргументите на конст. от базовия клас. Един конструктор може да наследи базов конструктор така:

<производен-конструктор>(<списък аргументи>):<базов-клас>(<списък аргументи>) {
* //тяло на конструктора на производния клас*
}

Полиморфизъм-способността на обекти, принадлежащи към различни класове(типове) да изпълняват метод извикан с еднакво име, всеки според подходящия начин.
Операторът overloading на цифрови оператори =,-,*,/ позволява полиморфична обработка на различни числени типове: int,double т.н. всеки, от които има различни диапазони и представяне.
Полиморфизмът се използва при наследяването на класове. При тях е много подходящо общите методи да бъдат с еднакви имена. Полиморфизмът не е overloading или overriding. Полиморфизмът касае само до приложение със специфично изпълнение(представяне) на интерфейс или по-общо базов клас. Методът overloading се отнася до методи с еднакво име, но различни сигнатути. Методът overriding(динамичен полиморфизъм) означава да се предостави нова реализация на дадения метод, различна от наследената от суперкласа реализация. Новата реализация в подкласа има същото име, същия брой и тип на параметрите и връща същия резултат като реализацията на метода в подкласа.
Пример: Методът Speak() за куче ще върне лае, а за прасе същият ще върне грухти. Кучето и прасето наследяват метода Speak от животно, но методите на подкласа override методи на базовия клас известно като overring полиморфизъм.

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); }
  };
```

Член функцията area() е декларирана като виртуална в базовия клас защото по-късно се предифинира в наследения клас.
Следователно virtual е разрешаването на члена от наследеният клас със същото име като в базвия клас, да бъде подходящо извикан когато чрез указател/референция се обръщаме към него като към базов клас. 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.
Основната разлика е, че в абстрактният клас ако най-малко за един от тези членове липсва имплементация, ние не можем да създадем инстанция(обект) от този клас.
Но клас ,от който не може да се инстанцира обект, не е изцяло неизползваем. Можем да създадем указатели и да ги използваме напълно във всички полиморфични “ситуации”. Пример:

```
myclass poly; //НЕ МОЖЕ
myclass *poly1=new Rectangle(10,5);
```

Първият ред няма да сработи защото незнае какво е тялото на абстрактните методи и не може инстанцира обект от класа. Иначе, указателите от този абстрактен клас, могат да сочат не-абстрактни обекти от наследените класове.
Виртуалните членове на абстрактните класове дават на C++ полиморфични характеристики, така че обектно-ориентираното програмиране да се използва пълноценно в големи проекти.

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 функции се постига полиморфизъм по време на изпълнение. За целта виртуалната функция трябва да бъде извикана посредством указател. Когато указател към базов клас сочи към обект от производен клас, съдържащ виртуална функция, и тази функция се извика посредством указател, C++ решава коя версия на функцията ще бъде извикана въз основа на типа на обекта, сочен от указателя. А това решение се взема по време на изпълнение.
Пример:

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

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

Copy-конструктор се нарича конструктор, който се вика когато се инициализира даден обект чрез друг:
myclass x=y; //y явно инициализира x
myclass x(y); //същото
И във двата случая се вика copy конструктор. Трябва се отбележи че myclass x=y; не е оператор за присвояване ами инициализиране, т.е. ако се предефинира оператора за присвояване =, няма да влияе в този случаи. Ако не сме дефинирали copy конструктор, компилаторът ни дефинира автоматично за нас.
Copy конструкторът има следният синтаксис:
<име-на-клас> (<име-на-клас> &<обект>)
{ ... }
В него се подава референтно обекта-източник, т.е. самият обект а не негово копие. Пример:
myclass::myclass (myclass &other)
{ … }
Макар че автоматично се създава copy конструктор е хубаво ние да дефинираме наш си тъй като автоматично създадения е съвсем елементарен и просто копира полетата от обекта-източник в нашия обект. В случаи че имаме указатели сочещи към обекти, то ще се копира само адреса и така ще имаме 2 инстанции имащи за поле указател сочещ на едно и също място в паметта. Това поражда опасност при освобождаване на тази памет. Този частен случаи се избягва чрез copy конструкторът в които ние може да дефинираме как да копира другият обект в нашият, а именно да създаде копие на обектите сочени с указател, и да ги запомниме в нашия обект.
Операторът за присвояване по подразбиране прави копие на всички променливи на обекта в дясно към тези на обекта в ляво - точно това е отразено и в пълното му име на английски, дадено по-горе. Понякога това копиране е просто това, което искаме и затова не е необходимо да презаредим оператора за присвояване. Но в някои случаи е необходимо да инициализираме някои от променливите (особено ако те са указатели), да променим някои от статичните променливи на класа и прочие подобни дейности, които не се осъществяват от оператора за присвояване по подразбиране, а трябва да бъдат зададени от програмиста. Тогава презареждането на този оператор е неизбежно.

```
X obj1, obj2;  // create two objects of class X
obj2.setobj()  // set up the second object
obj1 = obj2;   // assign second to first object
```

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. Това означава, че в тялото на функцията се работи с оригиналните променливи, а не с техни копия. Затова rswap() дава верен резултат. – non-const
2. Без проблеми може да се предаде голям обект от даден клас. Ако обектът се предава по стойност, той се копира в стека, при всяко извикване на функцията. Тази операция изисква много място и при рекурсивни функции води до препълване на стека. Освен това копирането забавя изпълнението. – const референция.
Ако използваме аргумент от тип reference, но не искаме да променяме стойността на фактическия параметър, можем да декларираме формалния параметър като const.

```
class X;
111
int foo( X& );
int bar( const X& x ) {
// Обектът x е деклариран като const, а
// параметърът на foo() не може да бъде const
return foo( x ); // грешка
}
```

x не може да се предаде като аргумент на функцията foo(), докато нейната сигнатура не се промени на const X& или X.

Ако аргументът от тип reference се отнася за стандартен тип, а съответният фактически параметър не е от същия стандартен тип, може да се получи неочакван резултат. Ще се генерира временна променлива, чийто тип съвпада с декларирания. Тази променлива получава стойността на фактическия аргумент. На функцията се предава временната
променлива.
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. Стандартните типове данни не работят добре със този механизъм за предаване на параметри. Когато типът на фактическия аргумент не може да се определи предварително, по-добре pass by reference да не се използва.
Декларация:
int *&p1; - p1 е адреса на указател към обект от тип int.

Една функция може да връща резултат от тип reference или указател. Това е удобно при работа с класове, защото обектите не се копират. Друга възможна причина за използване на тип reference като тип на резултата, е необходимостта функцията да стои отляво на оператора за присвояване. Този тип на резултата означава, че функцията връща адреса на обекта. Функция с такъв тип на резултата е операцията за индексиране на клас IntArray. Тя връща резултат от тип reference, защото трябва да позволява четене и запис на стойност в елемент на масива. Следва дефиницията на функцията и един пример за нейното използване:

```
int& IntArray::operator[]( int index ) {
return ia[ index ];
}
IntArray myArray( 8 );
myArray[ 2 ] = myArray[ 1 ] + myArray[ 0 ];
```

Функцията може да връща референция(еквивалентно на връщането на указател)
const double& XY::GetConstX() const {return x}; Функцията връща константна референция към ХУ обекта и се поставя само от дясната страна на декларацията.

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

```
class XY{
public:
double x,y;
XY() {x =0.0; y = 0.0;} //default
XY(double a, double b;) {x = a; y = b;} //explicit constructor
XY(const XY& xy) // copy constructor
{
    x = xy.x; y = xy.y;
}
const XY& operator=(const XY& xy) //assignment operator
 {   x = xy.x; y = xy.y;
return *this; } };
class Orbiter
{
protected:
double mass;
XY m_prior, m_current, m_thrust;
```

```
public:
Orbiter (XY& current, XY& prior, double mass)
    : m_current(current),m_prior(prior),m_mass(mass){} const XY& GetPosition() const;
void Fly();
virtual void Display() = 0;
};


class Planet : public Orbiter
{
public:
Planet(XY& current, XY& prior, double mass) : Orbiter(current, prior, mass) {}
void Display();
};

class SpaceShip : public Orbiter
{
private:
 double m_fuel;
XY m_orientation;
public:
SpaceShip( XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
: Orbiter(current, prior, mass) };
```

Конструиране на вградени обекти:

1) Компилаторът обработва декларацията на обекта и „знае“ точно колко памет е необходима за даден клас и я заделя;
2) Всички вградени обекти са конструирани;
3) Извиква се конструктора на родителския клас;
4) m_orientation е конструиран вграден обект;
5) функцията на конструктура на Spaceship се извиква

Деструкция на вградени обекти:
Как се разрушава класът Spaceship: той наследява класа Orbiter и има вградени обекти, които са дефинрани в родителския клас и в дъщерния. Следователно:

1) Извиква се деструктора на Spaceship;
2) m_orientation е вграден обект и се разрушава;
3) Извиква се деструкторът на Orbiter;
4) Вградените обекти m_current, m_prior и mass се разрушават;
5) Паметта, заемана от Spaceship е освободена;

Извикване на конструктор и деструктор на родителски и наследяващ клас:

```
// BaseClass declaration
class BaseClass {
public:
BaseClass() // Constructor
 { cout << "This is the BaseClass constructor.\n"; }
~BaseClass() // Destructor
{ cout << "This is the BaseClass destructor.\n"; } };

// DerivedClass declaration
class DerivedClass : public BaseClass {
public:
DerivedClass()  { cout << "This is the DerivedClass constructor.\n"; }
~DerivedClass()  { cout << "This is the DerivedClass destructor.\n"; } };
```

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 (free storage). Една особеност на динамичната памет е това, че тя не се свързва с име на променлива. С разположените в нея обекти се работи косвено чрез указатели. Втора особеност на динамичната памет е това, че тя е неинициализирана. Следователно, преди да я използваме, трябва да я инициализираме. Динамична памет се заделя от операцията new, приложена към стандартен тип или име на клас. В динамичната памет може да се задели място за отделен обект или за

масив от обекти. Например int *pi = new int; , заделя място за един обект от тип int. new връща указател към обекта и pi се инициализира с този указател.
IntArray *pia = new IntArray(1024);
заделя място за един обект от клас IntArray. Ако след името на класа има скоби, в тях стоят аргументи за конструктора на класа. Ако скобите липсват, класът трябва да притежава конструктор без аргументи или да не притежава конструктори. Ако след името на типа стои заграден в квадратни скоби израз, new заделя място за масив от обекти. Изразът може да бъде с произволна сложност и се нарича размерност. New връща указател към първия елемент на масива.

Може да се задели място и за масив, чиито елементи са обекти от даден клас. Например,
IntArray *pia = new IntArray[someSize];
Активност за една променлива - частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта.
Паметта за глобалните променливи се заделя в началото на програмата и остава свързана с тях до завършване на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането и. Статичните локални променливи имат период на активност като глобалните променливи.
Паметта за динамичните променливи се заделя от операцията new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на памет се извършва от операцията delete, приложена към указателя, който адресира съответната променлива.

```
void IntArray::grow()
{
int *oldia = ia;
int oldSize = size;
size += size/2 + 1;
ia = new int[ size ];
// копира елементите на стария масив в новия
125
for ( int i = 0; i < oldSize; ++i) ia[ i ] = oldia[ i ];
// инициализира останалите елементи с 0
for ( ; i < size; ++i ) ia[ i ] = 0;
delete oldia;
}
```

Ако се освобождава памет, заета от масив, чиито елементи са обекти от даден клас, трябва да се посочи дължината на масива. Тя е необходима, за да се определи броят на деструкторите, които ще се извикат.
Нека имаме следната дефиниция:
IntArray *pia = new IntArray[size];
Заетата от този масив памет се освобождава така: delete [size] pia;
delete трябва да се използва само за памет, заделена чрез new. В противен случай ефектът е непредсказуем. Няма забрана за прилагане на delete към указател със стойност 0. Ако стойността на указателя е 0, той е свободен и не адресира нищо.

Динамичната памет не е неограничена. Тя може да се изчерпи при изпълнение на програмата. Неуспешното завършване на delete ускорява изчерпването. new връща нулев указател, ако наличната в момента динамична памет е недостатъчна. Не бива да се пренебрегва възможността new да върне указател със стойност 0. В C++ съществува библиотека, която улеснява следенето на динамичната памет.
Един обект може да се разположи на точно определено място в динамичната памет. За целта се използва следния вид на операцията new:
new (адрес_на_мястото) спецификатор_на_тип;
където адрес_на_мястото е указател. За да се използва тази форма на операцията new, в програмата трябва да се включи заглавният файл new.h. Ако предварително заделим място в паметта, по-късно можем да разположим на това място конкретен обект, като използваме специалната форма на new.
Проблеми:

- Несъответствие в броя на заделянията и освобождаванията, което 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.
- Опит за писане в незаделена от програмиста памет на валиден адрес в адресното пространство на вашата програма или запис на повече информация от заделената за това памет – проблем, допринесъл за най-големите пробиви свързани със сигурността.

- Бавно заделяне (и освобождаване) на динамична памет;
- Неефективно използване на процесорите на машината поради неоптимизирани алгоритми за синхронизация на структурите от данни на мениджъра на паметта по подразбиране;
- броене на референциите към обектите, най-вече при йерархии от обекти, където има циклични референции.

  Съществуват свободни имплементации на Garbage Collector за C и C++, които обаче не са широко разпространени и използвани.

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

Приятелските функции на един клас се дефинират извън областта на действие на този клас, но имат право на достъп до закритите елементи (private) и защитените елементи (protected) на дадения клас. Функция или цял клас могат да бъдат обявени като приятели на друг клас. Приятелските функции се използват за повишаване на производителността.
Обектите от класа на итератора се използват за да избират последователно елементи или да изпълняват операции над елементите на обект от класа контейнер. Типични операции върху елементи на обект от клас контейнер са вмъкване, изключване, търсене, сортиране, проверка за наличие на елемент и др..
Примери за класове контейнери са масиви, стекове, опашки, свързани списъци.
За да се обяви, че функция е приятел на клас преди прототипа на функцията в описанието на класа се записва ключовата дума friend. За да се обяви например, че ClassTwo е приятел на ClassOne, трябва в описанието на ClassOne да се запише: friend ClassTwo;
В този случай всички функции елементи на ClassTwo стават приятелски функции за ClassOne. Обикновено класът на итераторите се обявява като приятелски на класа на контейнерите. Обявяване на приятелство може да стане на произволно място в описанието на класа.
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. Скриването на информация може да се използва и в този случай. Статичният член може да е скрит, докато глобалната променлива винаги е общодостъпна.
2. Статичният член не принадлежи на глобалната област на действие. Това премахва възможността за случайно използване на еднакви имена на променливи.
Член, който се използва за представяне на класа, става статичен, ако декларацията му започва с ключовата дума static. Достъпът до статичните данни на класа зависи от секцията, в която са дефинирани.
Статичните данни на даден клас се инициализират извън неговата дефиниция, както се инициализират променливи, които не са членове на клас. Ето как може да се инициализира
costPerShare:
#include "CoOp.h"
double CoOp::costPerShare = 23.99;
По подразбиране всички членове са по инстанция(т.е не састатични)
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. Следователно, тези инициализации не трябва да се включват в заглавен файл. Тяхното място е при дефинициите на тези функции-членове на съответния клас, които не са inline. Нивото на достъп до статичните данни на класа се отнася само за тяхното четене и промяна, но не и за инициализирането им. Обръщението към статичните методи може да се извърши чрез обект или указател към обект от класа, както се извършва обръщението към нестатичните методи. Обръщението може да се бъде и директно. Статичните методи на класа не

поддържат указателя this. Всяко явно или неявно използване на този указател ще предизвика грешка при компилация.

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

Предекларирane на оператор за даден клас означава да се състави функция, която да се изпълнява, когато един или повече обекти от този клас се подложат на съответния дефиниран оператор. Операторните декларации са разширение на съществуващите в езика оператори върху новосъздадените класове.
Операциите на езика C++ “=”, “+”, “-”, “*”, “/”,“+-”, “==” и други имат общоприето значение в синтаксиса на езика. При създаването на нови класове това семантично значение може да се пренесе върху обектите на класа и да се осъществява със съответната дефинирана от създателя на класа функция. Предефиниране на оператора е създадената от програмиста функция, която се изпълнява за (върху) обектите от новосъздадения клас.
Пример: Бинарната операция сума, която се използва при математически операции е предекларирана върху класа string, като се имплементира като съединяване (конкатенация) на обектите от класа (низове): string str1("aaa") ;
string str2("bbb") ;
string str3 = str1+str2; // aaabbb
За всеки от създадените класове могат да се предекларират общоизвестните или да се декларират напълно нови като символна последователност и начин на действие оператори.
Операторите (+, -, *) за сумиране, изваждане, умножение на обекти с числа или помежду им. За целта се дефинира функционалност, съответстваща на тези операции, обработваща член променливите на класа. Не могат да се предефинират символите: “.” (точка) достъп до член чрез име на обекта; “.*” (точка звезда) указател към член на обекта; “::” (двойно двуеточие) за определяне обхват; “?:” (въпросителна друеточие) условен оператор.
Правила:
-Не може да се използват други символи за предекларирane на съществуващите оператори. Използват се само тези, които се използват в C++;
-Предекларираният оператор запазва семантиката на действието си от този, който предекларира. Това означава, че асоциативността и приоритета на оператора ще се възприемат от този, който е предекларирan.
-Операторът не е комутативен, докато не се дефинира като такъв;
-Всички функции за предекларирane на оператори се наследяват с изключение на оператор за присвояване “=”;
-Всички оператори се дефинират самостоятелно (като отделен низ от символи). Например дефинирането на оператори “-” и “=” не се пренася автоматично върху оператора “-=”;
-Операторите не могат да се реализират като статични член функции, освен операторите new и delete които са статични за езика C++.

Унарни оператори
Могат да се предекларират операторите + и – . Синтаксис за оператор +:
ТипНаРезултата Тип::operator+();
където ТипНаРезултата е връщаната стойност от предекларираната функция на оператора +;
Тип е типът на операнда. Борави с членовете на класа.

Или: ТипНаРезултата operator+()(Тип); Борави с членовете на Тип.

Декларацията на типа може (и трябва) да има и атрибути, като референция & (и константна спецификация - const). При наличие на дефиниция-тяло на функцията, която се изпълнява при срещането на оператора може да се използва операцията върху обект от клас Тип. Пример: +ОбектОтКласаТип;

Бинарни оператори
Дефиниция Бинарните оператори са операторите, които свързват два аргумента. Биват два основни вида в съответствие с връщания резултат:
-бинарни аритметични оператори на езика (+, -, *, /)
-бинарни оператори за сравнение (,<=,>=,==,!=). Те се прилагат при изрази между обектите от даден клас, например: Обект1ОтКласаТип + Обект2ОтКласаТип;
Изчислението на изрази от тази форма се свежда до извикване на предефинираната функция, която се изпълнява. Функцията, предефиниращa оператора връща резултат, например обект: ОбектРезултатОтКласаТип

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;
double b = sin(a); //sin очаква double -> компилаторът го преобразува.
```

Когато обаче създаваме наши класове компилаторът незнае как да преобразува дадения клас в друг тип. Трябва ние сами да опишем метода по който ще става това „преобразуване”. Пример:

```
class MyString {
     char *s;
     int size;
   public:
     operator const char * () { return s; } //conversion operator
   ...
};

void main() {
   MyString str(“Hello”);
   strcmp(str, “Hello”);  //Тук се извиква оператора за преобразуване.
}
```

Ако операторът за конверсация липсваше, то когато се извика strcmp, която очаква char*, а му се подаде MyString str, компилаторът ще покаже грешка защото незнае как да преобразува MyString във char*. Слагайки този „метод”, той се извиква при нужда от превръщане в този тип. Особеност при този „метод” е че не се задава тип на връщаната стойност (пред оператор), и че винаги няма аргументи.

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. Действиво 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. Code Contracts bring the advantages of design-by-contract programming на всички .NET програмини езици. Съставени са от 3 основни tools: **Runtime Checking(програмата се модифицира чрез инжектиране на контракти, които са част от**

изпълнението на програмата), Static Checking(определя контрактни грешки преди стартирането на програмата) и Documentation Generation(увеличава броя на съществъващите XML файлове с инфорамция за контрактите.)

2) **Инварианти**

Описват условия, които следва да са винаги true за времето на която и да е инстанция на класа. Казано по друг начин, invariant указват условие, което следва да се поддържа през времето на всяко взаимодействие между класа и негов клиент — което значи при всяко изпълнение на public members, *включително и на constructors.*
Има някои условия, които очакваме да са верни за даден комплексен обект. Тези условия се наричат инварианти. По очевидна причина за тях се предполага, че внаги са верни. Един от начините за създаването на качествен код е да се открие кои инваранти са подходящи за съответните класове и да се напише код, който не допуска грешки в класовете.
Една от соновните причини, за които капсулирането на данни е полезно е прилагането на инварианти. Първата стъпка е да се предотврати неограничен достъп до инстанцираните променливи като се направят private. След това обектът се модифицира чрез accessor функции и modifiers. Ако прегледаме всички accessor- и и modifier-и, можем да видим, че всеки от тях управлява инварианти и следователно е невъзможно инвараинтът да има лошо влияние.

Пример:

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

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

3) Пред и пост- условия

Много често, когато създаваме функция можем да предположим параметрите, които тя ще върне. Ако тези предположения не са верни, то тогава програмата връща грешка. За да работи правилно една програма, много добра идея е да помислите върху предположенията си , да ги запишете като част от програмта и да напишете код, който да ги проверява. Например, следващата функция:

```
void Complex::calculateCartesian ()
// precondition: the current object contains valid polar coordinates
//  and the polar flag is set
// postcondition: the current object will contain valid Cartesian
//  coordinates and valid polar coordinates, and both the cartesian
//  flag and the polar flag will be set
{
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

Предполага се, че флагът polar е сетнат, а mag и theta съдържат валидни данни. Ако това не е вярно, то отгава функцията ще възпроизведе некоректни резултати.

Можем да добавим коментар към функцията, който да предупреждава програмистите за пред-условие.

```
// precondition: the current object contains valid polar coordinates
//  and the polar flag is set
// postcondition: the current object will contain valid Cartesian
//  coordinates and valid polar coordinates, and both the cartesian
//  flag and the polar flag will be set
```

В същото време, може да коментираме и post-улсовията – нещата които знаем ще са верни когато функцията се изпълни. Тези коментаари са полезни за хората, които ще четат вашите програми, но е още по-дорра идея да с добави код, който проверява пред-условията, за да може да се принтира подходящо съобщение за грешка:

```
void Complex::calculateCartesian ()
{
  if (polar == false) {
    cout <<
    "calculateCartesian failed because the polar representation is invalid"
     << endl;
    exit (1);
  }
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

Обобщение:
Пред-условието показва кое трябва да е вярно преди да се извика функцията;
Пост-условието показва какво трябва да е вярно, след като функцията се изпълни.

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

Още един пример:
```
bool is_vowel( char letter )
//   Precondition:  letter is an uppercase or
//   lowercase letter (in the range 'A' ... 'Z' or 'a' ... 'z') .
//   Postcondition:  The value returned by the
//   function is true if Letter is a vowel;
//   otherwise the value returned by the function is
//   false.
```

26. Контракт за инварианти – поглед в дълбочина. Реализация в .NET. Може да се ползва и горната тема за този въпрос.

Съществуват 2 контрактни инвариантни метода:

1) Invariant(Boolean)

```
[ConditionalAttribute(L"CONTRACTS_FULL")]
public:
static void Invariant(
            bool condition
)
```

2) Invariatn(Boolean,String)

```
[ConditionalAttribute(L"CONTRACTS_FULL")]
public: static void Invariant( bool condition, String^ userMessage)
```

Инвариантните контракти се съдържат в метод, определен от атрибута ContractInvariantMethodAttribute. Обикновено този метод се нарича ObjectInvariant.

- Тези контракти могат да се определят само в инвариантен метод, деклариран в класа;
- Не са отворени за клиенти и затова могат да реферират членове;
- Използва се двоично записване по време на изпълнение на инвариантите;
- Инвараинтите са условно дефинирани в **CONTRACTS FULL** symbol. По време на проверката на изпълнението, инвариантите се проверяват в края на всеки публичен метод. Ако инвариант посочва, публичен метод в същия клас, то тогава той се проверява само в края на най-външния метод, който е извикан в съответния клас.

The invariant can be checked with an **assert()** expression:

1. classes need to pass a class object

2. structs need to pass the address of an instance

```
auto mydate = new Date(); //class

auto s = S();            //struct

...

assert(mydate);          // check that class Date invariant holds

assert(&s);              // check that struct S invariant holds
```

27. 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.

```
public class Rectangle
{
  public virtual Int32 Width { get; set; }
  public virtual Int32 Height { get; set; }
  [ContractInvariantMethod]
  private void ObjectInvariant()
  {
    Contract.Invariant(Width > 0);
    Contract.Invariant(Height > 0);
  }
}public class Square : Rectangle
{
  public Square()
  {
  }
  public Square(Int32 size)
  {
    Width = size;
    Height = size;
  }
  [ContractInvariantMethod]
  private void ObjectInvariant()
```

```
  {
    Contract.Invariant(Width == Height);
  }
  ...
}
```

Базовият клас Rectangle има 2 инварианта : width и height >0. Дъщерният клас Square добавя още едно инвариантно условие: width и height да бъдат равни. Дъщерният клас е като базовия с изключение на това, че има допълнително ограничение: width и height трябва винаги да бъдат равни. За пост-условията нещата за по същия начин. Дъщерният клас предефинира/пренаписава даден метод и добавя още пост-условия, за да се увеличат възможностите на базовия клас и да работи като специален случай на базовия клас, който извършва допълнителни функции, освен основните.
А какво става с пред-условията? Обобщаването на контрактите сред йерархията на класовете е деликатна операция. Логически, методът на класа наподобява математическа функция – и двете имат входни стойности и съответни изходни стойности. В математиката всяка функция има интервал от стойности - областта на допустимите стойности.

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

1) SOLID принципи: съвкупност от : Принцип на едноличната отговорност -Single Responsibility Principle (SRP); Отворено-затворен принцип - Open Closed Principle (OCP); Принцип на субституцията (Лисков) - Liskov's Substitution Principle (LSP); Принцип за разделяне на интерфейсите - Interface Segregation Principle (ISP); Принцип за обръщане на зависимостта- Dependency Inversion Principle (DIP)).

С - Всеки клас трябва да има една причина за съществуване
О - Отворен за разширение, затворен за промени
Л - Лисковият Принцип за заместване на единици от програмата
И - Интерфейсите трябва да не бъдат дебели
Д - Д ... Единици функционалност трябва да зависят от абстрактни единици, не една от друга. Абстрактни единици не трябва да зависят от Детайли около една единица, Детаилите около единицата трябва да зависят от Абстракции.

2) Open/closed принцип: Един модул трябва да бъде отворен за разширение, но затворен за модификация.

Ако има нови изисквания към софтуера, тогава няма да модифицираме вече работещия код, а ще имплементираме нов.
Класовете, модулите, функциите и т.н, трябва да са отворени за разширение, но затворени за модификация. Чрез абстракции може да се промени поведението на дъщерния клас. С други думи, създавайки базовия клас с предефируеми функции имаме възможност да създаваме нови класове, които изпълняват същата функция по различен начин, но без да се променя базовата функционалност. Ако свойствата на абстрактния клас трябва да се сравнят или организират да работят заедно, трябва да се добави още една абстракция, която да се спарави с това.

3) Принцип на регламентираната отговорност:
Един модул трябва да има само една причина да се променя. Този принцип гласи, че ако имаме две неща да се променят в един клас, трябва да се разделят функционалност в два класа. Всеки клас ще се справи само с една отговорност и за бъдеще, ако ние трябва да направим една промяна, ние ще го направи в класа, в който се отнася.

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

Какво ще стане ако полиморфизмът не е достатъчен за обработка на всички възможни комбинации? Може да се използва принципът на двойния dispatch, за да изтегли комбинацията в подкласове по такъв начин, че да не се „счупят“ интерфейсните контракти.
Double Dispatch е техника, която се използва в контекста на полиморфизма и се извиква, за да „смекчи“ липсата от поддръжка на мултиметод в програмните езици. По-просто, Double Dispatch се използва , за да извиква предефиниран метод, в който параметрите ваарират сред йерархията на наследяването.
Всеки тип в йерархията има специален код, с който взаимодейства с други типове от йерархията. Подходът за различно представяне на поведението на даден клас, създава код, който е не чак толкова гъвкав към бъдещи промени и в същото време е по-труден за разширяване(донаписване).

31. 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. Когаато 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.

Допълнение:
Функции които ползват указатели/референции към базов клас, трябва да могат да могат да ползват обекти от наследени на този клас, без да знаят за това:

```
Animal *ptr = new Cat(…);
ptr = new Dog(…);
ptr->speak();
```

В този случай ще се извика speak() на класът Animal, освен ако той не е виртуален. Ако е, ще се извика методът на класът Dog наследяващ класа Animal, ако го има. Това понякога поражда проблеми и за това е препоръчително наследяващия клас само да разширява базовия клас, без да изменя неговата функционалност/поведение.
Пример: square класът наследява rectangle класът като припокрива ВИРТУАЛНИТЕ функции:

```
void square::setWidth(int w) { width = w; height = w; }
void square::setWidth(int w) { width = h; height = h; }
```

След което ако се извиква следната функция ше се види грешка в логиката:

```
void f(rectangle *ptr)
{
  ptr->setWidth(5);
  ptr->setHeight(10);
  ptr->printArea(); //width*height
}
```

Ако към функцията се подаде rectangle инстанция, всичко ще бъде наред и ще излезе 50. Но ако подадем square, ще върне 100, което не е очаквания резултат, гледайки го като rectangle.

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

1) Асинхронни задачи:

```
int a, b, c;
int calculateA() { return a+a*b; }
int calculateB() { return a*(a+a*(a+1)); }
int calculateC() { return b*(b+1)-b; }
int main(int argc, char *argv[])
{
  getUserData(); // initializes a and b
```

```
c = calculateA() * (calculateB() + calculateC());
showResult(); }
```

Главната функция пита потребителя да въведе данни и след това се изпълняват 3-те функции: calculateA, calculateB и calculateC. Аритметичните функции са програмирани така, че през интервал от 1 до 3 сек. се изпълнява всяка.Тъй като тези стъпки се изпълняват последователно това води до общо време за изпълнение 9сек(след като данните са въведени) Функциите са независими и затова можем да ги изпълним паралелно чрез функцията async:

```
int main(int argc, char *argv[])
{
 getUserData();
future<int> f1 = async(calculateB), f2 = async(calculateC);
c = calculateA() * (f1.get() + f2.get());
showResult();
}
```

В горния пример се използват async и future. И двете са дефинирани в std namespace. Future приема функция, ламбада или обект от функция и връща future. Тази концепция наподобява контейнер за евентуален резултат. Кой резултат? Този, който е върнат от извиканата функция асинхронно. Ще се нуждаем и от резултатите на паралелните функции. Извиквайки гет-метода във всеки future, блокираме изпълнението докато стойността не стане достъпна. В най-лошия случай, изчакването на тази модификация е около 3 секунди, за разлика от 9-те секунди при последователното извикване на функциите.

2) Нишки

Моделът на асинхронната задача, предсатвен в предишната точка може да ни е полезен при някои сценарии, но ако се нуждаете от по-дълбока обработка и контрол над изпълнението на нишки, C++11 се включва с thread класа, деклариран в <thread> header и намиращ се в std namespace.Нишките предлагат по-добри методи за синхронизация и координация, позволявайки да предават изпълнението на задачата към друга нишка и да изчакват определен интервал от време или докато друга нишка завърши. В следващия пример има ламбада функция с целочислен аргумент.

```
auto multiple_finder = [](int n) {
   for (int i = 0; i < 100000; i++)
   if (i%n==0)
 cout << i << " is a multiple of " << n << endl; };
 int main(int argc, char *argv[])
 { thread th(multiple_finder, 23456);
  multiple_finder(34567);
  th.join();
}
```

Предаваме ламбада към нишката: функция или обект от нея. В мейна изпълняваме функцията чрез 2 нишки с различни параметри.

3) Thread-bound променливи и изключения

В C++ може да дефинираме глобалните променливи, чийто обхват граничи с приложението, включващо нишки. Но спрямо нишките, съществува начин за определяне на тези глобални променливи , така че всяка нишка да пази свое копие. Тази концепция е позната като thread local storage и се декларира по следния начин:

```
thread_local int subtotal = 0;
```

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

Желателно е всички програми да се разделят на напълно независими отделни части от асинхронната задача. В практиката това почти никога не е възможно, тъй като има зависимости между данните и всички части ги изпълняват едновременно. C++11 използва следните технологии, за да избегне конкурентните условия:

1) Атомични типове – подобни на примитивните типове, но позволяват thread-safe модификация. <atomic> header включва серии от примитивни типове: atomic_char, atomic_int. По този начин тези типове са еквивалентни на техните едноименни без префикса atomic_, но с разликата, че всички техни присвоени оператори(==,++,--,+=,*= и т.н) са предпазени от конкурентни условия. В следващия пример има 2 паралелни нишки(едната е главна), които търсят различни елементи с един и същи вектор:

```
atomic_uint total_iterations;
vector<unsigned> v;
int main(int argc, char *argv[])
{ total_iterations = 0;
v scramble_vector(1000);
thread th(find_element, 0);
find_element(100);
th.join();
cout << total_iterations << " total iterations." << endl; }

void find_element(unsigned element)
{ unsigned iterations = 0;
find_if(begin(v), end(v), [=, &iterations](const unsigned i) -> bool { ++iterations;
return (i==element);
});
total_iterations+= iterations;
 cout << "Thread #" << this_thread::get_id() << ": found after " << iterations << " iterations." << endl; }
```

2) Мутекси и заключения – елементи, които ни дават възможност да дефинираме thread-safe критични области;

<mutex> header дефинира серии от заключващи се класове, за да определят критичните области. По този начин, може да дефинираме мутекс,за да установим критична област от край до край в сериите от функции или методи - само 1 нишка в даден момент може да достъпи член от тези серии, заключвайки своя мутекс.

Нишка, която се опитва да заключи мутекс може да остане блокирана докато мутекса е достъпен или просто да се провали нейния опит. Алтернативният timed_mutex class може да бъде блокиран за малък интервал от време преди да се провали.

Заключеният мутекс трябва да се отключи за другите, които ще го заключват. Ако не успее, това може да доведе до неопределено поведение на програмата – податлива на грешки. Най-лошо е когато не се извърши заключване, защото това означава, е програмата не може да функционира правилно ако друг код продължава да чака този лок. За щастите C++11 притежава заключващи се класове.

```
mutex mx;
void funcA();
void funcB();
int main()
{ thread th(funcA); funcB();
th.join();
}
```

Този мутекс се използва, за да се гарантира, че 2-те функции A и B могат да се изпълняват паралелно, без да се събират в критичната област. Функция A може да изчака преди да достигне критичната област. За да я накараме да изчаква трябва да използваме най-простия механизъм – lock_guard

```
void funcA()
{ for (int i = 0; i<3; ++i)
{ this_thread::sleep_for(chrono::seconds(1));
cout << this_thread::get_id() << ": locking with wait... " << endl;
lock_guard<mutex> lg(mx);
... // прави нещо в критичната област. Принтира “lock secured”
cout << this_thread::get_id() << ": releasing lock." << endl;
}
}
```

По начина по който е дефинирана функция A, тя трябва да достъпи критичната област 3 пъти. Вместо това, Функция B ще опита да заключи, но ако той вече е заключен, тя ще изчака още секунда, докато опита отново да достъпи критичната област. Механизмът, който използва е unique_lock с try_to_lock_t:

```
void funcB() {
int successful_attempts = 0;
for (int i = 0; i<5; ++i)
{ unique_lock<mutex> ul(mx, try_to_lock_t());
if (ul) {
++successful_attempts;
```

```
cout << this_thread::get_id() << ": lock attempt successful." << endl; ... // Do something in the critical region cout
<< this_thread::get_id() << ": releasing lock." << endl;
}
else { cout << this_thread::get_id() << ": lock attempt unsuccessful. Hibernating..." << endl;
this_thread::sleep_for(chrono::seconds(1)); } } cout << this_thread::get_id() << ": " << successful_attempts << "
successful attempts." << endl;
}
```

По начина, по който е дефинирана функция B, тя ще опита 5 пъти да достъпи критичната област.

3) Условни променливи – начин за замръзяване на нишки от изпълненое докато не се изпълни даден критерии.

Producer() функция зарежда елементи в опашка:

```
mutex mq;
condition_variable cv;
        queue<int> q;
        void producer()
        { for (int i = 0;i<3;++i) { ... // Produce element
        cout << "Producer: element " << i << " queued." << endl;
    mq.lock();
    q.push(i);
    mq.unlock();
    cv.notify_all();
    } }
```

Стандартната опашка не е thread-safe, затова трябва да се уверим, че никой друг не я използва, когато елементите се зареждат.Функцията Consumer опитва да извлече елементи от опашката, когато е достъпна или изчаква известно време в условна променлива преди да опита пак. След 2 последователни неуспешни опита, функцията приключва:

```
void consumer() {
unique_lock<mutex> l(mq);
int failed_attempts = 0;
while (true) {
mq.lock();
 if (q.size())
{
 int elem = q.front();
q.pop();
mq.unlock();
 failed_attempts = 0;
cout << "Consumer: fetching " << elem << " from queue." << endl; }
else {
mq.unlock();
if (++failed_attempts>1) { cout << "Consumer: two consecutive failed attempts -> Exiting." << endl;
break; }
else { cout << "Consumer: queue not ready -> going to sleep." << endl; cv.wait_for(l, chrono::seconds(5)); } } } }
```

34. 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. Базиран е на
Типовете Task и Task<TResult> в System.Threading.Tasks namespace представя случайни асинхронни операции.
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()
{ // For this example, we're just going to (asynchronously) wait 100ms.
```

await Task.Delay(100); }

“async” активира “await” в този метод.Това е всичко което async прави!
Отнесено към метод или ламбда–израз ги изпълнява (разбира се, след повикване) в рамките на същата нишка,в която е викащия метод.
Началото на async- метода се изпълнява както всеки един метод. Стартира синхронно докато активира “await” (или хвърля exception).

Думата “await” е там където нещата се достъпват асинхронно. Await е като унарен оператор – приема един аргумент(който е awaitable – асинхронна операция). Await преглежда awaitable, за да провери дали е изпълнен. Ако е изпълнен, тогава методът продължава да се изпълнява(синхронно както обикновен метод), а ако не е изпълнен, тогава Await действа асинхронно. Казва на awaitable да стартира останалата част от метода когато се изпълнии след това се връща от async-метода. Когато awaitable е изпълнен, той ще изпълни останалата част от async-метода.

2) Awaitable

“await” приема 1 аргумент – awaitable, който е асинхронна операцоя. Има 2 типа awaitable:

- Task<T>
- Task One

Едно важно нещо за Awaitable: той е типът, който е Awaitable, а не методът, който връща тип. С други думи, можем да очакваме резултата от асинхронен метод, който връща Task … , защото методът връща Task, а не защото е async. Следователно може да очакваме резултат от синхронен метод, който връща Task:

```
public async Task NewStuffAsync()
{
// Use await and have fun with the new stuff.
await ...
}
public Task MyOldTaskParallelLibraryCode()
{
// Note that this is not an async method, so we can't use await in here.
...
}
public async Task ComposeAsync()
{
// We can await Tasks, regardless of where they come from.
await NewStuffAsync();
await MyOldTaskParallelLibraryCode();
}
```

3) Резултат от асинхронна операция.

Async методите могат да връщат Task<T>, Task, void. В повечето случаи бихме искали да върне Task<T> или Task. Защо? Защото те са Awaitable. Ако имаме асинхронен метод, който връща Task<T> или Task можем да предадем резултата към изчакване (await). С void метод, не няма какво да предадем към изчакване. Асинхронните методи, които връщат Task или void нямат връщана стойност. Методите, които връщат Task<T>, трябва да върнат стойност от тип T.

```
public async Task<int> CalculateAnswer()
{ await Task.Delay(100); // (Probably should be longer...)
// Return a type of "int", not "Task<int>"
return 42; }
```

Когато декларираме метод, който връща един тип и трябва да върне друг тип:

```
public async Task<int> GetValue()
{
await TaskEx.Delay(100);
return 13; // Return type is "int", not "Task<int>"
}
```

Async методи, които връщат стойност трябва да имат следния тип на резултата: Task<TResult>.

Има разлика между async void и async Task:
- async Task методът е както другите асинхронни операции, само че не връща стойност;
- async void методът действа като „top-level” асинхронна операция.
- async Task може да бъде композиран в друг асинхронен метод, използващ await;
-async void може да борави със събития.

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

1) Превключване на контексти

Когато очакваме изпълнението на awaitable, awaitable улавя текущия „контекст“ и след това го прилага към остатъка от асинхронния метод. Какво точно е „контекст“? Ако сме на UI нишка, то тогава това е UI контекст. Ако отговаряме на ASP.NET request, то това е ASP.NET request контекст**.**

```
// ASP.NET example
protected async void MyButton_Click(object sender, EventArgs e)
{ // we asynchronously wait, the ASP.NET thread is not blocked by the file download.
// This allows the thread to handle other requests while we're waiting.
await DownloadFileAsync(...);
 // Since we resume on the ASP.NET context, we can access the current request.
// We may actually be on another *thread*, but we have the same ASP.NET request
// context. Response.Write("File downloaded!"); }
```

Избягване на контекстите:
През по-голямата част от времето не е нужно да синхронизираме обратно към “main” контекста. Най-често асинхронните методи се проектират така: те очакват други операции и всяка една представлява асинхронна операция. В този случай, искаме да кажем на awaiter-а да не улавя текущия контекст, като извикваме ConfigureAwait. Пример:

```
private async Task DownloadFileAsync(string fileName)
{
// Use HttpClient or whatever to download the file contents.
var fileContents = await DownloadFileContentsAsync(fileName).ConfigureAwait(false);
// because of the ConfigureAwait(false), we are not on the original context here.
// Instead, we're running on the thread pool.
// Write the file contents out to a disk file.
await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);
```

2) Async композиция

Възможно е да се стартират няколко операции и да се изчакват докато една или всички не се изпълнят. Можем да реализираме това като стартираме операциите, но не ги изчакваме докато не стане нужно:

```
public async Task DoOperationsConcurrentlyAsync()
{ Task[] tasks = new Task[3];
tasks[0] = DoOperation0Async();
tasks[1] = DoOperation1Async();
tasks[2] = DoOperation2Async();
// At this point, all three tasks are running at the same time.
// Now, we await them all.
await Task.WhenAll(tasks);
}
public async Task<int> GetFirstToRespondAsync()
{
// Call two web services; take the first response.
Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };
// Await for the first one to respond.
Task<int> firstTask = await Task.WhenAny(tasks);
// Return the result.
return await firstTask;
```

	Description
await task	**Wait/await for a task to complete**

await task	**Get the result of a completed task**
await Task.WhenAny	**Wait/await for one of a collection of tasks to complete**
await Task.WhenAll	**Wait/await for every one of a collection of tasks to complete**
await Task.Delay	**Wait/await for a period of time**
Task.Run or TaskFactory.StartNew	**Create a code-based task**

36. 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’.
-Нека създадем генетичен списък: List<T> (произнася се : List of Tee):

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{       public List();
        public void Add(T item);
        public void Sort( IComparer<T> comparer);
        public T[] ToArray(0);
        ….
        public Int32 Count {get;}
        …
}
```

Начин на обработка:

За да се поддържат генетични имплементации, към .NET се добавиха:
1.Нови IL инструкции, четящи конкретния тип на аргумента;
2.Метаданновото описание се обогатява с описание на типа на параметрите;
3.Променя се синтаксисът на C#, Visual Basic и т.н.
4.Променят се компилаторите;
5.Променя се JIT компилаторът, така че да генерира ‘native code’ за всяко повикване с конкретен тип на аргумент.

Разлика с шаблонизирани типове

-Разработчикът не е нужно да притежава сорса на генеричния алгоритъм (за разлика от C++ templates или Java generics) за да прекомпилира.CLR средата генерира ‘native code’ за всеки метод, първият път когато методът
се повика с указан тип данни. Това разбира се, увеличава размера на кода (при генерични реализации), но не и производителността
При шаблоните, компилаторът генерира separate source-code functions (named specializations) при всяко отделно повикване на ф-ия шаблон или инстанция на шаблонизиран клас.

-ясен код: рядко се налагат tape casts;
-Подобрена производителност: преди генетиците, същото се постигаше с използване на Object типа. Това налага непрекъснато пакетиране (boxing), което изисква памет и ресурс, форсира често включване на с-мата за garbage collection. При генетичните алгоритми няма пакетиране.Това подобрява десетки пъти производителността.

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

Open & Closed types

Тип с генетични параметри се нарича ‘open type’ тъй като не допуска CLR да конструира инстанции директно (както е и при интерфейсите) Когато кодът се обърне към генетичен тип, се подават реални параметри. Тогава типът се нарича вече ‘ closed type’ и за него се прави инстанция.

Generic types and Inheritance

Това си е нормален тип и наследяемост е напълно допустима.

```
internal sealed class Node<T> {
        public T m_data; public Node<T> m_next;
        public Node(T data) : this(data,nul) {}
        public Node(T data, Node<T> next) {
                m_data = data; m_next = next; } ….
                                }
```

Използваме в производен тип:

```
private static void SameDataLinkedList() {
        Node<Char> head = new Node<Char>(‘C’);
        head = new Node<Char>(‘B’, head);
        head = new Node<Char>(‘A’, head);
                                }
```

Подменяне на генетични типове С цел удобство, е честа практика:
ако имаме: List<DateTime> dtl = new List<DateTime>();
да предефинираме: internal sealed class DateTimeList : List<DateTime> {}
И тогава можем да създадем списък от генетичен тип по традиционния начин:

```
DateTimeList dtl = new DateTimeList();
```

Обработка на генетични типове: code explosion

- При повикване на метод от генетичвен тип, JIT компилаторът прави заместването и създава ‘ native code’ за точно този метод с точно тези подменени параметри.
-CLR генерира native code за всеки метод/тип комбинация. Това води до ‘code explosion’.
-Ako впоследствие, метод се повика със същия тип аргумент, не се генерира повторен код.

-Еднократно се генерира и код в случаите, когато сргументите са от референтен тип. Напр: List<String> List<Stream> макар и аргументите всъщност да сочат съвсем различни неща.

При наследяване от негенетичен към генетичен интерфейс, трябва да се вътрешно пакетиране (преобразуване) на аргументите(boxing), Което е загуба на ресурс и бързо действие

Ето един стандартен в FCL интерфейс:

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
    T Current { get;}
    }
```

Ето клас, който имплементира горния интерфейс над тип Point:

```
internal sealed class Triangle : IEnumerator<Point> {
    private Point[] m_vetrices;
    public Point Current { get {… }
    }
```

-Ограничители (в генетични типове)
– constraints чрез тях може да се ограничи броя на типовете, които могат да са заместители в аргументите на генетичен тип:

```
public static T Min<T>(T o1, Y o2) where T : IComparable
    {
     if(o1.CompareTo(o2) < 0) return o1;
     return o2;
    }
```

38. 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.

39. Обекти в паметта – особености при разполагането и чести програмни грешки
В C обектите в паметта се разполагат с командите:

- calloc()
- malloc()
- realloc()

Паметта се освобождава с функцията free()
В C++ за разполагане на обекти в паметта може да се използва и оператора new, а за освобождаване на паметта се използва оператора delete. В C++ може да се използват и командите от C.
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.
realloc (void *p, size_t size);
*Променя големината на блока с памет към който сочи p към размерни битове.
* Ново локализираната памет е неинициализирана.
* Ако p е NULL, командата е еквивалента на malloc(size)
*Ако size е 0, командата е еквивалентна на free(p).
calloc(size_t nmemb, size_t size);
*Локализира памет за масив от nmemb елементи с големина size и връща пойнтер към локализираната памет.
*Паметта се нулира.
Чести грешки: Повечето C 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.

40. 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. пре-използва освободените (free) парчета (chunks) памет със най-малки загуби. При освобождаване се сращват 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}. [size] е размерът на {data}, парчето. Примерно malloc(8) ще задели 8 байта + [size] и ще върне адреса на парчето в паметта, където ще пише [8]{…}. Ако сме заделили повече парчета памети ще изглеждат така: [8]{…} [32]{……} [4]{..} [128]{……}, долепени едно до друго (незадължително). Така ако си на първото парче и искаш да стигнеш до 3-то парче ще скочиш един път 8 байта + още 32 байта надясно и ще стъпиш на 3-то парче.
Във [size] се включва и допълнителен бит - PREV_INUSE накрая, който посочва дали ПРЕДИШНОТО парче е освободено в момента.
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]. По средата се намира unused секция с неизползваема информация, останала от старите данни, може и да липсва. Структурата на едно освободено парче памет приема следният вид:
free
Forward и back указателите сочат следващото/предишното свободно парче. Пази се списък от освободените парчета, а не запазените. Когато се опита да се алокира ново парче, започва да се обхожда списъкът докато не се намери подходящо свободно парче, то се откъсва от списъка и върху което да алокира желаната памет. Използвайки това че

size 1
data

size 1
forward free
back
unused
size(същия)

са списък, допълнителния бит PREV_INUSE и повтарящия се [size] накрая на освободените парчета, може да се направи лесна дефрагментация на паметта, и две съседни свободни парчета да бъдат слепени.
Пример: [8]{…} [32]{…free…} [8]{free} [4]{..} [8]{free} ще бъдат открити че стоят заедно и ще бъдат сляти: [8]{…} [40]{…free…} [4]{..} [8]{free}.

42. 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 пойнтер може да сочи към първата извикана деструктор функция(нейния адрес може да бъде намерен в сектора dtors на програмата). Недоброжелателят може да намери този адрес и да се опита да го замести с пойнтер сочещ към негова функция. Когато second се освободи frontlink() започва да го слепва към fifth блок. Резултата е че във forward пойнтера на fifth е записана адрес който сочи към функция и при извикването на return(0) вместо деструктор функция ще се извика друга предоставена от недоброжелателя.

41 и 43. Препълване на буфер – опасности и използване за недобросъвестно вмъкване на код. Опасности при двойно освобождаване на памет.

Препълване на буфер се нарича, 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;
}
```

Забележка: един char символ е 1Byte! Низът „hello” е от 5 char символи5 Byte подред.
Когато заделим first който е по голямо парче- 666 байта, очакваме че при заделянето на second ще бъдат заделени непосредствено след first, тъй като е малко парче:
[?]{…}**[666]**{……..}**[12]**{…} **[?]**{…}
Когато се извика **free**(second); това парче бива „освободено” и вкарано в свързаният списък с свободни парчета. В самото парче старите данни {data} биват пренаписани със служебна информация като forward pointer и back pointer.
Така подредени, се изпълнява strcpy(first, BADSTR), която взима низът от втория аргумент и го копира на първия, без никакви проверки. В такъв случаи ако сме заделили 666 байта за first, а подадем низ BADSTR с размер 668, то ще бъдат пренаписани 668 байта от началото на first надясно т.е. 2 байта ще бъдат пренаписани върху следващата клетка, а именно **[12]** полето ще бъде засегнато. Ако запишеме повече байтове и {...} полето ще бъде засегнато. Тъй като second парчето е вече освободено, в него пише служебна информация- указатели. Така може да пренапишем {…} полето така че да пренасочим указателите към други адреси.

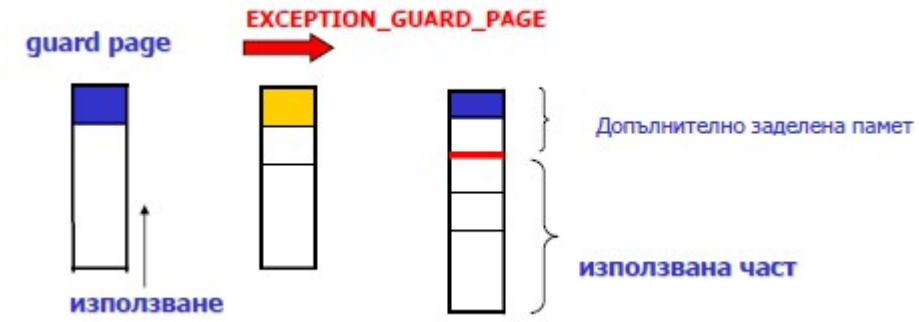
По-подобни начини може да бъде изменена/повредена структората на паметта, да бъдат пренасочени собствени функции към функциии изпълняващи зловреден код или най-малкото да променим действието на програмата. Такива атаки може да използват **unlink**, **frontlink**, двойно освобождаване на паметта и т.н. Дефрагментацията и слепването също спомагат за това.
Повторно освобождаване на вече освободена памет води до объркване на логиката на списъците, което също е опасно.

44. Динамично управление на памет в Windows.

HeapAlloc() – за големи блокове
VirtualAlloc() – за малки блокове

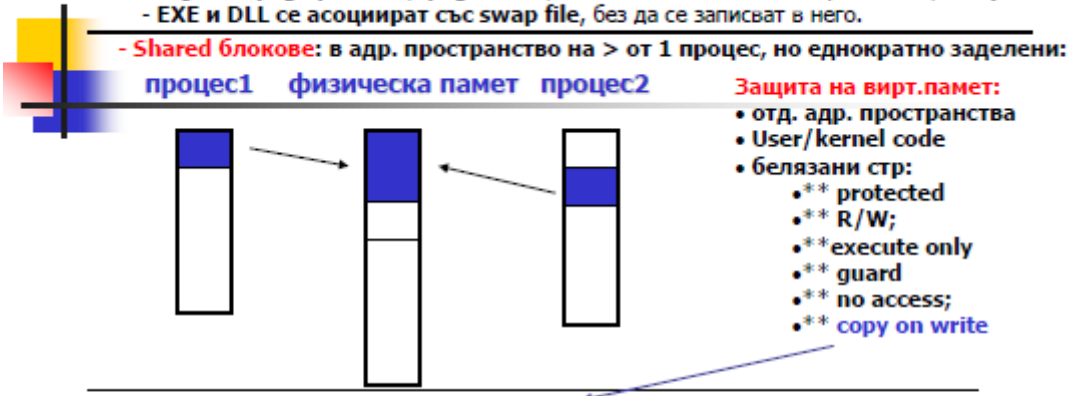
Заделя се блок с определена големина в рамките на нуждите процеса. Този блок не може да се резервира повторно.
pMem = VirtualAlloc(<нач.адрес на блока или NULL>,<брой стр. за резервиране>,MEM_Reserve,<права за достъп>);

Заделянето става по страници(например 4K) и по необходимост, в ОП и swap file от резервираната, след което може да се използва паметта. След изчерпването ѝ се генерира exeption: exeption_guard_page.



Елементи на виртуалната организация на паметта:

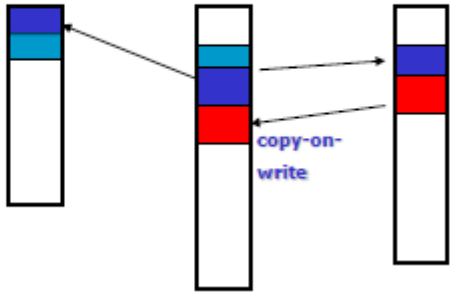
- **4GB къмпроцес, разделени на страници;** CR3→points 1page table dir (1024 page-tables) →1 page-table points 1024 pages
- flags for: '**page present**'; '**page fault**'; динамично сваляне на стр. на диск; **swap file.**
- **EXE и DLL се асоциират със swap file**, без да се записват в него.
- **Shared блокове: в адр. пространство на > от 1 процес, но еднократно заделени:**



Защита на вирт.памет:

- **отд. адр. пространства**
- **User/kernel code**
- **белязани стр:**
  - ** protected
  - ** R/W;
  - **execute only
  - ** guard
  - ** no access;
  - ** copy on write

Поделяне на глобални данни м/ду процеси: **отделни копия и отложено размножаване**



Lazy evaluation

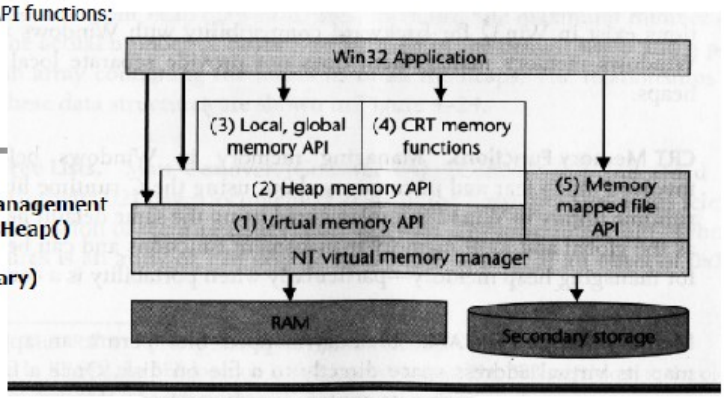
45. Служебни структури в динамичния менижмънт на паметта в ОС Windows.

Windows memory management (with RtlHeap)

RtlHeap is the memory manager on Windows. Uses API functions for memory management

5 sets of Windows API functions:

1. 4K pages, reserved, committed, page management
2. HeapCreate(), process heap, GetProcessHeap()
3. Only for compatibility with old versions
4. In Win32 environment (C Run-time Library)
5. Discussed later



Win32 memory management APIs

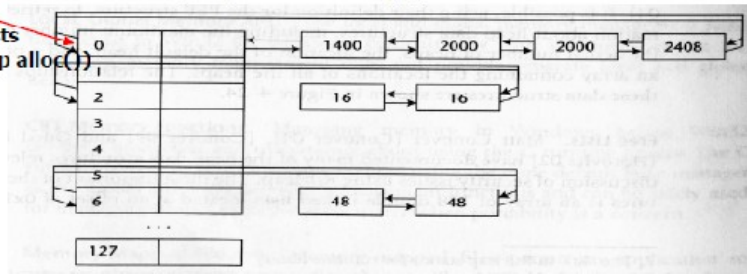
RtlHeap data structures:

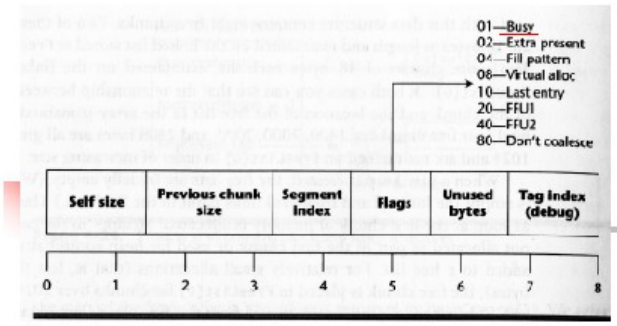
-Process environment block(PEB) –info about internal data structures, number and addresses of heaps

-Free lists: located at 0x178 from the start of the heap (HeapCreate()). Used to keep track of free chunks Contains 128 double linked lists for chunks of the same size (exception is FreeList[0] containing buffers >1024 bytes)

-look-Aside lists: up to 128 single linked lists for small memory blocks (<1K)-to speed up alloc()

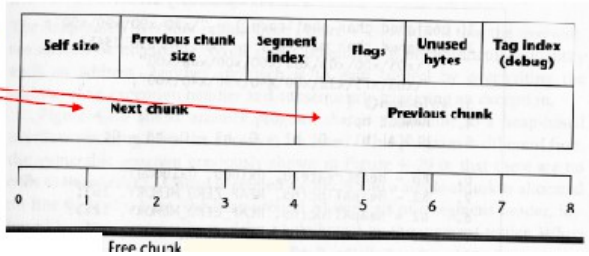
-Memory chunks: a control structure associated with each allocated chunk by HeapAlloc() or malloc(). The structure precedes the address returned by 8 bytes.





Control structure (for 1 memory chunk) associated with each allocated by heapAlloc() or malloc() chunk (all chunks are multiples of 8)

After free() or HeapFree() memory is added to the coresponding free list with index for memory chunks of that size (see prev. slide)
Pointers point to free lists of same size or to the head of the list.
Memory is left in its place.



46. Препълване на буфер в Windows и атаки, базирани на това. Пример. Техники за

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;
 ...
 }
```

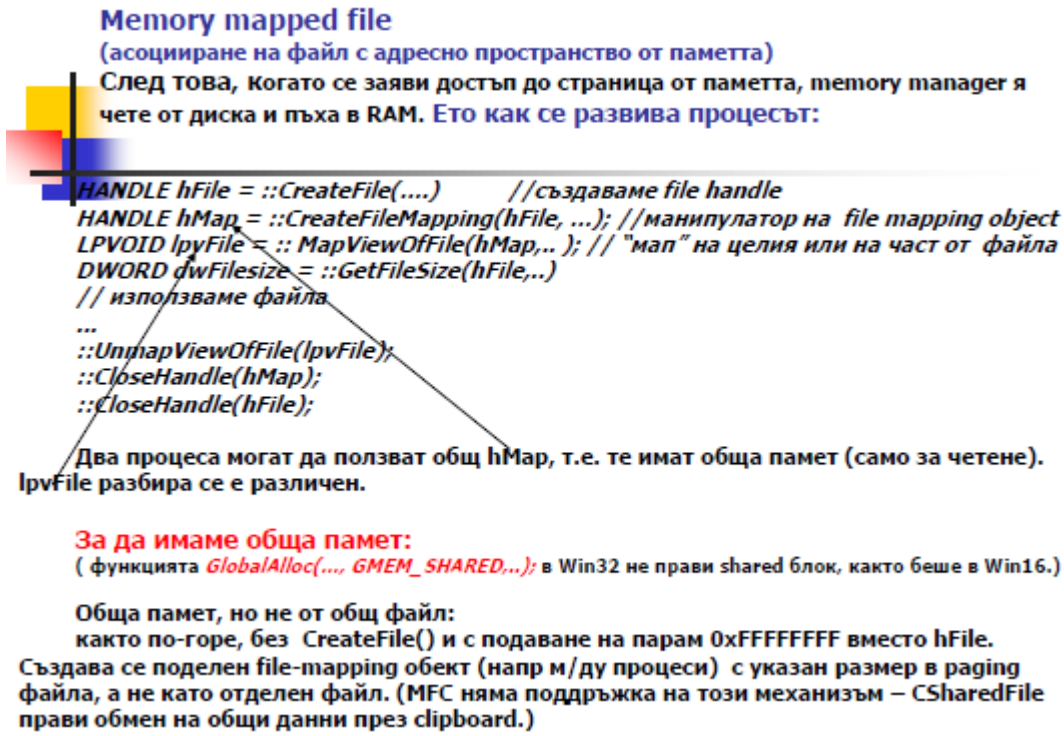
Горната програма е валидна и всеки компилатор би я компилирал без да даде съобщение за грешка. Обаче програмата се опитва да пише отвъд заделената за буфера памет, което може да доведе до неочаквано поведение от нейна страна.

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 харддиска 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-a (тоест адреса, където трябва да се отиде след изпълнение на функцията), че даже и данните за параметрите на функцията. По този начин се „повреждa” 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 A-та, шестнайсетичната стойност на която е 0x41, което означава че return address-a 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.

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



* 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