

## 25. CASE: обектно-структурирана програмна система - плащане. UML диаграми на класовете, дефиниране достъп, методи, пропъртита, права. Полиморфизъм при изпълнението и преобразувания над типове.

Фразата "CASE: Обектно-структурирана софтуерна система - Плащане" изглежда като кратко описание или заглавие, което се отнася до конкретна софтуерна система или проект. Ето разяснение:

**\*\*CASE:\*\*** Този акроним обикновено означава "Computer-Aided Software Engineering" (Инженеринг на софтуер с помощта на компютър). Това се отнася до използването на компютърни инструменти за помощ при разработката и поддръжката на софтуерни системи. Тези инструменти могат да включват графично моделиране, генериране на код и поддръжка на документацията.

**\*\*Обектно-структурирана софтуерна система:\*\*** Това показва, че софтуерната система, за която става въпрос, е проектирана и структурирана въз основа на принципите на обектно-ориентираното програмиране. Обектно-ориентираното програмиране (ООП) включва организирането на софтуера като колекция от обекти, всеки представляващ инстанция на клас и взаимодействащ с останалите.

**\*\*Плащане:\*\*** Това подсказва, че фокусът или основната функционалност на тази софтуерна система е свързана с плащания. Това може да включва обработката на финансови транзакции, управлението на начини за плащане или обработката на различни аспекти на операциите, свързани с плащанията.

## 26. Разширяване функционалността на системата за плащания: въвеждане на интерфейси. Полиморфична употреба на методи, наследяване на интерфейс.

**Разширяване функционалността на системата за плащания:** Това означава добавяне на нови възможности или подобрения в системата за обработка на плащания. Тези промени могат да включват нови функционалности, подобрена ефективност или разширено покритие на видове транзакции.

**Въвеждане на интерфейси:** Въвеждането на интерфейси обикновено се отнася до дефиниране на абстрактни структури, които указват какви методи трябва да бъдат реализирани от класовете, които ги използват. Интерфейсите могат да помогнат за стандартизацията и облекчаването на взаимодействието между различни компоненти на системата.

**Полиморфична употреба на методи:** Полиморфизмът се отнася до способността на обект да предоставя различни реализации на методите в зависимост от контекста. Този принцип може да улесни използването на обекти с различни типове, без да се изисква предварително познаване на техните конкретни типове.

## 27. Често използвани интерфейси в .NET.

В .NET (Microsoft's framework for building applications), съществуват няколко често използвани интерфейса, които предоставят стандартни функционалности и служат за определени цели в програмирането на приложения. Някои от тези интерфейси включват:

1. **`IEnumerable`** и **`IEnumerator`**: Тези интерфейси се използват за реализиране на итерация чрез обекти, които могат да бъдат изброени. Те са основа за `foreach` цикъла и предоставят възможност за последователно преминаване през елементите на колекция.
2. **`IComparable`** и **`IComparer`**: Тези интерфейси позволяват на обектите да бъдат сравнявани, което е полезно за сортиране на данни в колекции.
3. **`IDisposable`**: Този интерфейс дефинира метод за освобождаване на ресурси, като например затваряне на файлове или освобождаване на мрежови връзки. Обикновено се използва с ключовата дума `using``.
4. **`INotifyPropertyChanged`**: Този интерфейс се използва в паттерна на проектиране "Observer" и позволява на обекти да известяват своите наблюдатели (например UI компоненти) за промени в техните свойства.
5. **`IQueryable`**: Този интерфейс се използва в LINQ (Language-Integrated Query) и позволява изпълнението на заявки към различни видове данни, като например бази от данни.
6. **`IServiceProvider`** и **`IServiceCollection`**: Тези интерфейси се използват в ASP.NET и други части на .NET за инжектиране на зависимости и управление на услуги (services).

Тези са само някои от многото интерфейси, които се използват в .NET framework. Всякакъв обект, който имплементира даден интерфейс, обикновено трябва да предостави конкретна реализация на методите, декларирани в този интерфейс.

## 28. \*Софтуерни контракти. Пред и пост-условия. Инварианти.Примери.

Във .NET Framework 4, Code Contracts е framework създаден по-подходящ синтаксис за да се изразят класове и функции с множество изходни точки. Code Contracts поддържа 3 типа contracts:

- 1.Preconditions,
- 2.Postconditions,
- 3.invariants

**Preconditions** се занимават с предусловия, които следва да бъдат проверени за да може метод да се изпълни.

**Postconditions** се занимават с условия, които следва да бъдат проверени в момента, когато метод е завършил изпълнението си – коректно или с хвърлено изключение.

**Invariants** 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.

Code Contracts API се състои от static methods на клас Contract. методът Requires() се ползва за preconditions , а Ensures() за postconditions.

```
using System.Diagnostics.Contracts;
public class Calculator
{
    public Int32 Sum(Int32 x, Int32 y)
    {
        Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
        Contract.Ensures(Contract.Result<Int32>() >= 0);
        if (x == y)
            return 2 * x;
        return x + y;
    }
    public Int32 Divide(Int32 x, Int32 y)
    { Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
      Contract.Requires<ArgumentOutOfRangeException>(y > 0);
      Contract.Ensures(Contract.Result<Int32>() >= 0);
      return x / y;
    }
}
```

Въведен е т. нар . Code Contracts rewriter, който пробразува кода на етап компилация след анализ на целта на preconditions или postconditions. Той разширява автоматично кода и поставя новогенерираните блокове там, където им е мястото. Това означава, че разработчикът не се грижи къде да постави postcondition и дали ги е дублирал някъде в кода (особено при добавяне на нова exit point ).

*Синтаксис:*

При preconditions изразът съдържа input parameters и възможно е и друг method или property от същия клас. Към метода следва да се добави и атрибут 'Pure' за да се отбележи, че няма да се променят данни. Properties (getters) се подразбира че са pure.

При postconditions обикновено се реферира и друга информация, като например връщана стойност, или начална стойност на local variable. Това става с конструкции :  
Contract.Result<T> ..... - за да провери стойност (от тип T) връщана от метод и  
Contract.OldValue<T> ..... - за да вземе стойност (съхранявана в специална променлива) от началото на изпълнението на метода. Има възможност да се провери и условието в момент на генериране на exception (ако това стане) по време на изпълнение на метод.  
Това става с Contract.EnsuresOnThrow<TException> ....

## 29. \*Контракт за инварианти – поглед в дълбочина. Реализация в .NET.

Инвариантът е условие, което винаги е истина в обкръжението на определен контекст. Отнесено към ООП – условието следва да е истина за всяка инстанция на класа. Инвариантният контракт за даден клас е колекция от условия, които се задържат истина за периода на съществуване на инстанцията на класа.

- Инвариантният контракт се дефинира чрез 1 или повече методи.
- Те са private, void и с атрибут (както ще видим в примера).
- не съдържат друг код , освен условието за проверка.

```
public class News {  
    public String Title {get; set;}  
    public String Body {get; set;}  
    [ContractInvariantMethod]  
    private void ObjectInvariant()  
    {  
        Contract.Invariant(!String.IsNullOrEmpty(Title));  
        Contract.Invariant(!String.IsNullOrEmpty(Body));  
    }  
}
```

Инвариантите се проверяват в края на public методите. Но в тялото, временно статусът им може да стане невалиден. С инварианти можете да следите само преди и след изпълнение на public метод. Има отделен MS Static Code Checker който следи за присвоявания в тялото противоречащи на инвариантните ограничения

## 30. \*Контракт и наследяване. Проблем с ограничаване на областта в дъщерен обект.

Когато създаден контракт реализиран с private метод - pure (непроменящ се), се създава се метод, чиито наследник може да дефинира за да опише своите собствени инварианти.

```
public abstract class DomainObject  
{ public abstract Boolean IsValid();  
  [Pure] private Boolean IsValidState()  
  {  
    return IsValid();  
  }  
  [ContractInvariantMethod] private void ObjectInvariant()  
  {  
    Contract.Invariant(IsValidState());  
  }  
}
```

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

Принципите на SOLID представляват набор от основни принципи за обектно-ориентиран дизайн, които имат за цел да направят софтуерните системи по-гъвкави, устойчиви на промени и лесни за поддържане. SOLID е акроним за следните принципи:

Single Responsibility Principle - SRP:

Класът трябва да има само една причина за промяна. Това означава, че един клас трябва да извършва само една основна функционалност и да бъде отговорен само за това.

(Open/Closed Principle - OCP):

Софтуерният модул (клас, функция, модул и т.н.) трябва да бъде отворен за разширение, но затворен за модификация. Това означава, че добавянето на нова функционалност трябва да бъде възможно чрез добавяне на нов код, а не промяна на съществуващия.

Liskov Substitution Principle - LSP:

Обекти от суперкласа трябва да могат да бъдат заменени с обекти от подкласа без да се нарушава правилната работа на програмата. Този принцип стои зад понятието за "подтипове", където подкласът може да бъде използван вместо суперкласа без промяна на програмата.

Dependency Inversion Principle - DIP:

Високо ниво модули не трябва да зависят от ниско ниво модули. И двата трябва да зависят от абстракции. Абстракции не трябва да зависят от детайли. Детайлите трябва да зависят от абстракции.

Interface Segregation Principle - ISP:

Един клас не трябва да бъде принуден да имплементира интерфейси, които не използва. Този принцип насърчава разделянето на големи интерфейси на по-малки и специфични интерфейси, които се прилагат само към конкретни класове.

## 32. Обектен дизайн : принцип на верижната отговорност,

Принципът на верижната отговорност (Chain of Responsibility) е един от принципите на обектно-ориентирания дизайн, който насърчава построяването на верижка от обекти, където всеки обект в веригата има възможността да обработва заявката, както и опцията да я предава на следващия обект в веригата.

Принцип на верижната отговорност включва следните основни идеи:

Цел на дизайна:

- a. Разделянето на отговорностите между различни обекти и създаването на верига от тези обекти. Този принцип позволява на заявките да преминават през веригата от обекти, като всяко ниво има възможността да обработва или предава заявката на следващия обект в веригата.

Структура на веригата:

- b. Обектите са организирани във верига, където всеки обект има връзка със следващия обект в веригата. Това обикновено се реализира чрез наследяване или агрегация.

Обработка на заявката:

- c. Всеки обект в веригата решава дали да обработи заявката или да я предаде на следващия обект в веригата. Този избор може да зависи от вътрешното състояние на обекта, неговите способности и др.

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

Принципът на двойния dispatch (double dispatch) е дизайн паттерн, който се използва в обектно-ориентираното програмиране, за да се постигне гъвкавост и разширяемост във връзка със заявки (заявки за извикване) върху обекти. Този принцип позволява на програмата да избира метода за извикване въз основа на типа на два обекта (не само на един).

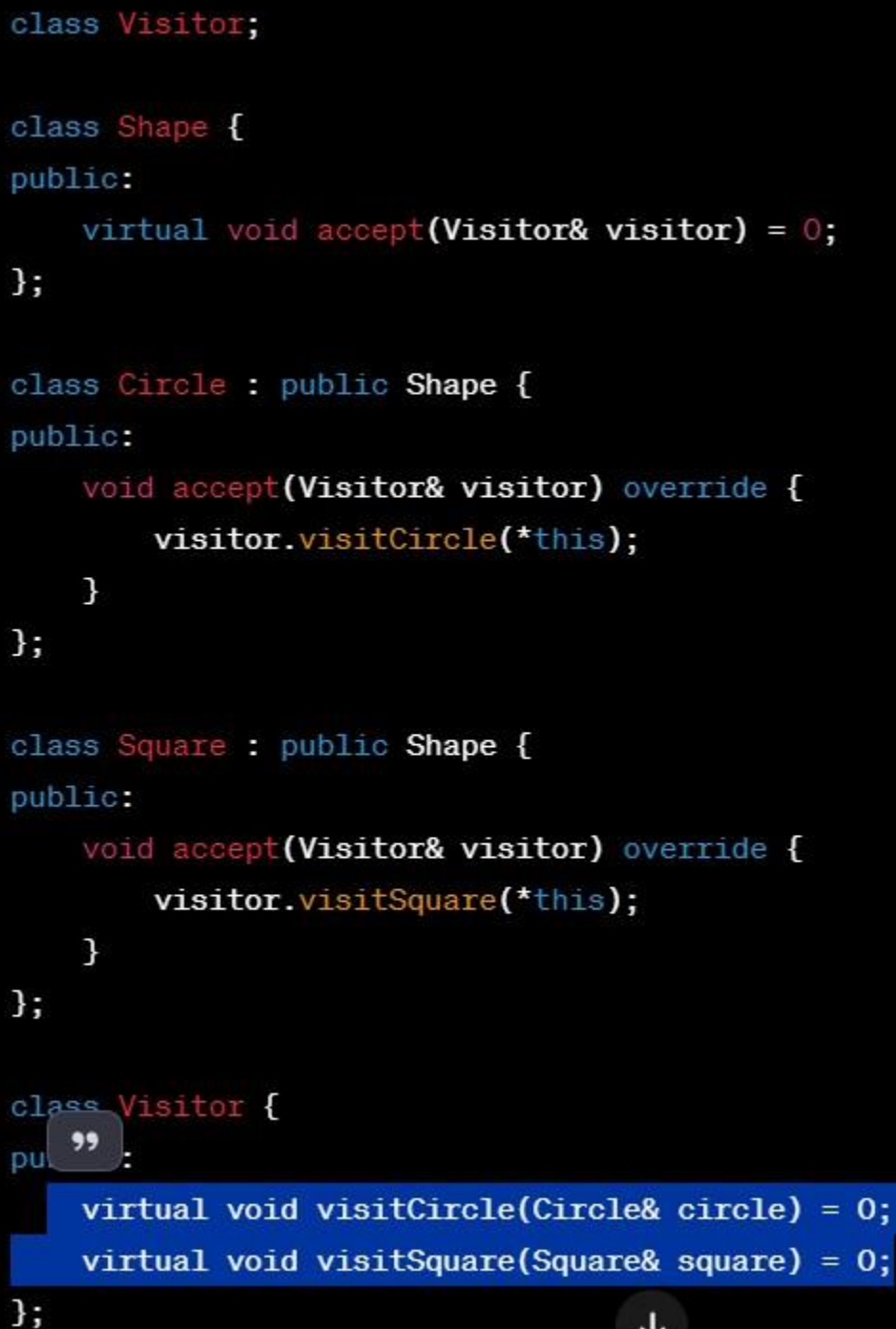
```
class Visitor;

class Shape {
public:
    virtual void accept(Visitor& visitor) = 0;
};

class Circle : public Shape {
public:
    void accept(Visitor& visitor) override {
        visitor.visitCircle(*this);
    }
};

class Square : public Shape {
public:
    void accept(Visitor& visitor) override {
        visitor.visitSquare(*this);
    }
};

class Visitor {
public:
    virtual void visitCircle(Circle& circle) = 0;
    virtual void visitSquare(Square& square) = 0;
};
```



```

class AreaCalculator : public Visitor {
public:
    void visitCircle(Circle& circle) override {
        std::cout << "Calculating area of Circle.\n";
        // Изчисления за площ на кръг...
    }

    void visitSquare(Square& square) override {
        std::cout << "Calculating area of Square.\n";
        // Изчисления за площ на квадрат...
    }
};

int main() {
    Circle circle;
    Square square;

    AreaCalculator areaCalculator;

    // Двойно разпознаване във виртуална таблица
    circle.accept(areaCalculator);
    square.accept(areaCalculator);

```

В този пример, класът Visitor представлява интерфейс за посещаване на различни типове обекти от йерархията Shape. Класовете Circle и Square имплементират метода accept, който извиква съответния метод на посетителя (Visitor). Когато методът accept се извика, обектът от йерархията Shape пренасочва извикването към подходящия метод на посетителя.

Програмата може лесно да се разшири с нови форми, като просто добавя нов клас към йерархията и имплементира съответния метод accept.

Принципът на двойния dispatch позволява изборът на метод да зависи от типа на два обекта, което улеснява добавянето на нови видове обекти или нови операции без промяна на съществуващия код.



## 34. Обектен дизайн : принцип на Лисков.

### Принцип на Лисков и контрактите в .NET.

Принципът на Лисков (Liskov Substitution Principle - LSP) е един от принципите на SOLID и е формулиран от Барбара Лисков. Този принцип дефинира условията, които клас трябва да спазва, за да може да замени свой базов клас, без да променя коректността на програмата. Принципът на Лисков утвърждава, че обекти от базовия клас трябва да могат да бъдат заместени от обекти на неговите подкласове без да се нарушава правилното функциониране на програмата.

Формално формулиран принципът на Лисков казва: "Ако S е подтип на T, тогава обектите от тип T могат да бъдат заместени с обекти от тип S без да се нарушава правилното функциониране на програмата."

```
class Shape {
public:
    virtual void draw() const {
        // изобразява формата
    }
};

class Circle : public Shape {
public:
    void draw() const override {
        // изобразява кръг
    }
};

class Square : public Shape {
public:
    void draw() const override {
        // изобразява квадрат
    }
};
```

Този код следва принципа на Лисков, тъй като Circle и Square са подкласове на Shape, и всички те могат бъдат използвани вместо Shape без проблеми

## 35. Генетични (пораждащи) типове в обектното програмиране. Синтаксис. Начин на обработка в .NET. 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.Метаданното 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.

## 36. 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,null) {}  
    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 генетичен 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'.
- Ако впоследствие, метод се повика със същия тип аргумент, не се генерира повторен код.
- Еднократно се генерира и код в случаите, когато аргументите 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), Кое 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;  
}
```

## 37. Lambda- изрази. Синтаксис и използване на ламбда-изразите. Функции-обекти.

Ламбда-изразите в C++ представляват анонимни функции, които могат да бъдат дефинирани на място и се използват за кратко и удобно представяне на функционалност. Те често се използват вместо функции или функтори, когато са нужни по място. Ламбда-изразите са особено полезни при работа с функционални стилове на програмиране или когато трябва да предадете функционалност на място.

```
[capture](parameters) -> return_type {
```

```
    // тяло на ламбда-израза
```

```
}
```

- Capture (Захващане): Определя какви променливи от външната среда ще бъдат видими в ламбда-израза. Може да бъде [=] (захванати са всички променливи по стойност) или [&] (захванати са всички променливи по референция), както и други комбинации.
- Parameters (Параметри): Списък с параметри на функцията, която ще се дефинира.
- Return Type (Тип на връщане): Типът на връщане на ламбда-израза.
- Тяло на ламбда-израза: Блок от операции, които се изпълняват, когато се извика ламбда-израза.

```
int main() {  
    // Ламбда-израз с параметри и връщане на стойност  
    auto add = [](int a, int b) -> int {  
        return a + b;  
    };  
  
    // Извикване на ламбда-израза  
    int result = add(3, 4);  
    std::cout << "Result: " << result << std::endl;  
  
    return 0;  
}
```

Ламбда-изразите в C++ се компилират до функции-обекти, които могат да бъдат създавани и присвоявани на променливи. Функции-обекти (функтори) са обекти, които могат да бъдат извиквани като функции. Те са полезни, когато искате да предадете функционалност като параметър или когато искате да съхраните състояние във функция.

cpp

Copy

```
#include <iostream>

// Пример за функция-обект (функтор)
struct AddFunctor {
    int operator()(int a, int b) const {
        return a + b;
    }
};

int main() {
    // Използване на функция-обект
    AddFunctor addFunctor;
    int result = addFunctor(3, 4);

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

## 38. Обекти в паметта – особености при разполагането и 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 поинтер 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 влияе на производителността и не е винаги необходимо. По – добрия вариант за локализиране на памет е с `memset()` или извикването на `calloc()`, което нулира паметта.

## 39. Управление на памет в конзолен режим и в Linux системи: служебни структури в паметта. Освобождаване на обекти (макрос `unlink()`).

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 срастват 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`  
`forwardfree`  
`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}`



## 40. Препълване на буфер. Поглед отвътре. Техника на вмъкване на код. Пример.

Препълването на буфер (buffer overflow) е вид атака, при която злонамерен код успява да запише повече данни в буфер от този, за който е предвиден. Този тип атаки често води до промяна в паметта и може да бъде злоупотребен за изпълнение на зловреден код. Една от техниките, свързани с препълването на буфер, е вмъкването на код (code injection).

### Техника на вмъкване на код (Code Injection):

Техниката на вмъкване на код включва вмъкване на изпълним код в областта на паметта на програмата, където той не е предвиден да се намира. Обикновено целта на тази техника е изпълнение на зловреден код от атакуващия.

```
void vulnerableFunction(const char* input) {
    char buffer[16];
    strcpy(buffer, input); // Потенциална точка за препълване на буфера
}

int main() {
    const char* maliciousInput = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                  "\x90\x90\x90\x90\x90\x90\x90\x90" /
                                  "\xCC\xCC\xCC\xCC\xCC\xCC\xCC\xCC";

    vulnerableFunction(maliciousInput);

    return 0;
}
```

Този пример илюстрира проста ситуация, при която функцията `vulnerableFunction` използва `strcpy` за копиране на входен низ в статично заделен буфер с фиксиран размер. Ако входният низ е по-дълъг от очаквания размер на буфера, може да се получи препълване на буфера. В този контекст, техниката на вмъкването на код може да бъде използвана за вмъкване на зловреден код в буфера с цел злоупотреба с изпълнението на програмата/

## 41. Атаката от тип - 'frontlink'. 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*argv[]){
    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 адрес към фалшив блок.
    strcpy(fourth, argv[1]);
    free(second);
    return(0);
}
```

Във фалшивия блок е съхранен адрес към поинтер на функция. Този поинтер 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 замени с поинтер сочещ към негова функция. Когато `second` се освободи `frontlink()` започва да го слепва към `fifth` блок. Резултата е че във `forward` поинтера 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.

## 42. Опасности при двойно освобождаване на памет (double-free vulnerabilities). Пример.

Двойното освобождаване на памет (Double-Free Vulnerabilities) е вид атака, при която злонамерен код успява да освободи едно и също блокче памет два пъти. Този вид атаки могат да доведат до сериозни проблеми в програмите, като например срив на програмата или злоупотреба с паметта.

```

int main() {
    // Създаване на блок памет
    int* ptr = new int;

    // Първо освобождаване на паметта
    delete ptr;

    // Отново освобождаване на същата памет (двойно освобождаване)
    delete ptr;

    return 0;
}

```

В този пример се опитваме да освободим същата памет два пъти с оператора delete. Това може да доведе до неопределено поведение на програмата. В някои случаи двойното освобождаване може да доведе до злоупотреба с паметта или дори да отвори врати за изпълнение на злонамерен код.

За предотвратяване на този вид атаки и проблеми с паметта, е добра практика след освобождаване на паметта да се установи нейната стойност на nullptr, което предотвратява случайното или злонамерено освобождаване на същата памет повторно

```

// Отново освобождаване на паметта, но този път с проверка за nullptr
if (ptr != nullptr) {
    delete ptr;
}

```

## 43. Динамично управление на памет в Windows.

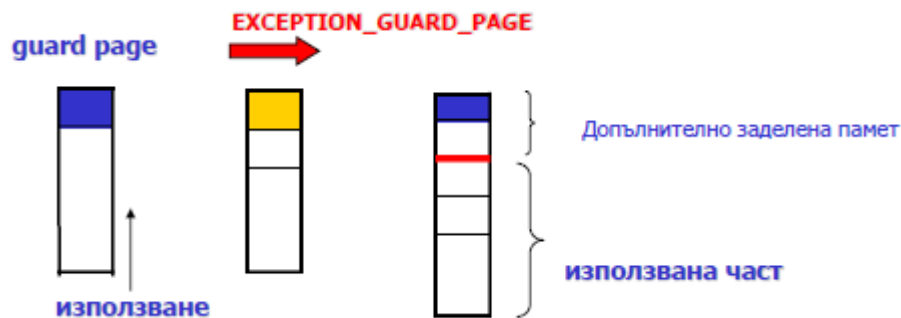
**HeapAlloc()** – за големи блокове

**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 exception: `exception_guard_page`.

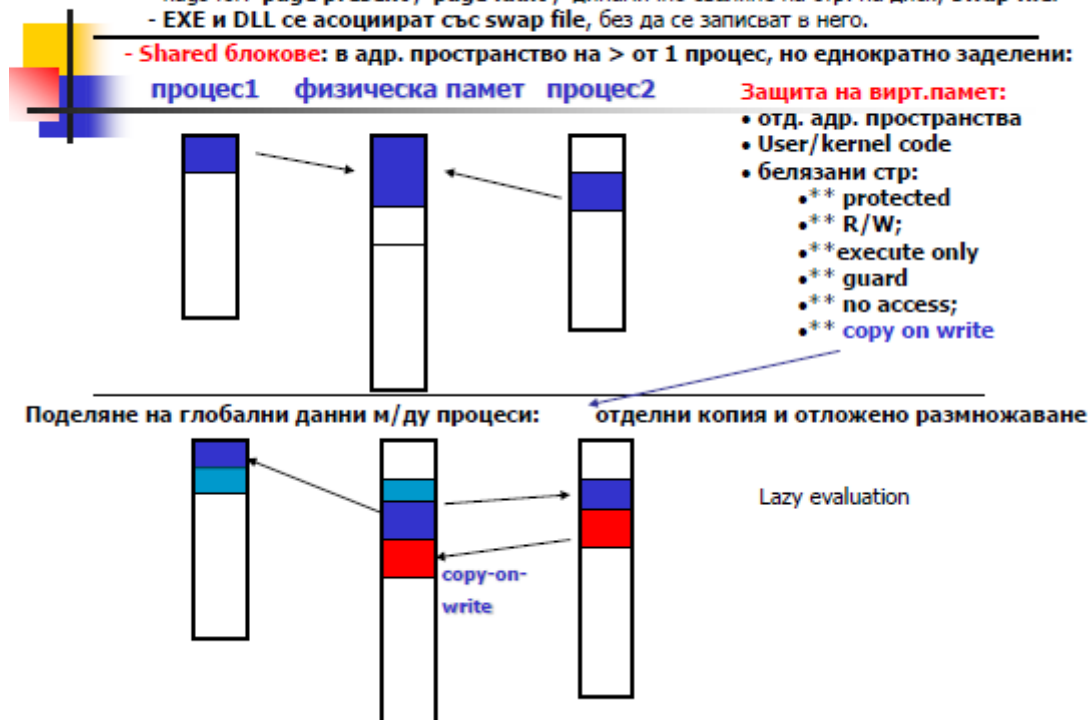


**Елементи на виртуалната организация на паметта:**

- 4GB към процес, разделени на страници; CR3 → points 1 page 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:

процес1    физическа памет    процес2



## 44. Служебни структури в динамичния мениджмънт на паметта в ОС Windows.

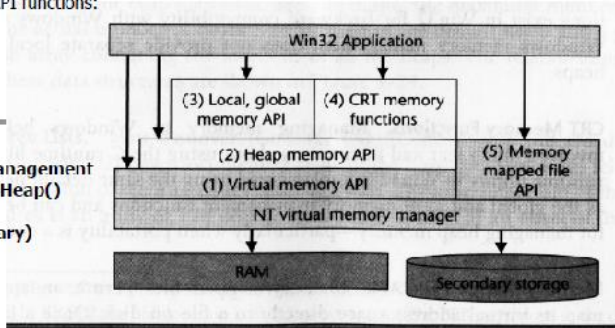
### Windows memory management (with RtlHeap)

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

5 sets of Windows API functions:

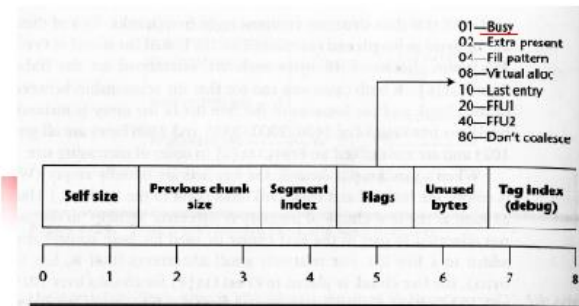
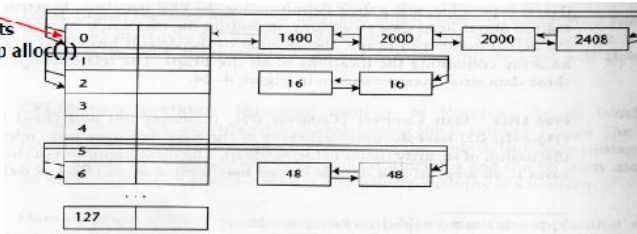


1. 4K pages, reserved, committed, page management
2. HeapCreate(), process heap, GetProcessHeap()
3. Only for compatibility with old versions
4. In Win32 environment (C Run-time Library)
5. Discussed later



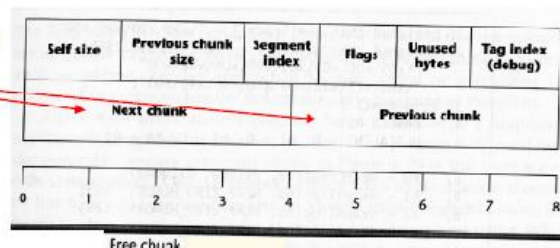
#### RtlHeap data structures:

- 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 corresponding **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.



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

Буферното препълване под windows се получава като се промени информацията в forward или backward поинтерите използвани в двустранно свързан листа. Това променя нормалното изпълнение на програмата към което може да се добави недоброжелателен код. За да може да се изпълни overflow-а ни трябва адрес който да е изпълним, откриването на този адрес е трудно, но възможно. Друг начин е да се получи достъп до адрес експетшън, който да бъде заменен.

## 46. \*Съпоставяне на файл с оперативна памет. Програмни практики при управление на паметта.

**Memory mapped file**  
(асоцииране на файл с адресно пространство от паметта)  
След това, когато се заяви достъп до страница от паметта, memory manager я чете от диска и пъха в RAM. Ето как се развива процесът:

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

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

**За да имаме обща памет:**  
(функцията `GlobalAlloc(..., GMEM_SHARED,...)`; в Win32 не прави shared блок, както беше в Win16.)

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

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

\* има 2 вида heap: 1. авт. заделен от ОС за приложението (`GlobalAlloc()`), която вика `HeapAlloc()`, или по-лесно – работи с `malloc/free`, или още по-лесно – `new/delete` и 2. собствени heap блокове:

= създаване	<code>hHeap = HeapCreate(..., размер);</code>
// може синхронизиран достъп до хипа от повече от 1 thread в рамките на процес	
= заделяне памет от създаден	<code>pHeap = HeapAlloc(hHeap, опции, размер);</code>
= освобождаване	<code>HeapFree();</code>

Някои съвети при работа със собствен heap

- \* Създавайте локален heap в рамките на своите класове (по 1 за клас)
  - \*\* избягва се се фрагментацията при продължителна работа.
  - \*\* нараства безопасността, поради изолацията в рамки на процес
  - \*\* позволява модифициране на `new`, `delete` операторите, конкретно за клас, в рамките 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`