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

Класовете са типове дефинирани от потребителя. Всеки клас съдържа данни и набор от функции, обработващи тези данни. Данните, които са компоненти на класа, се наричат данни-елемнти, функциите, които са компоненти на класа, се наричат функции-елементи. Подобно на това, както същност от тип int се нарича променлива, то същност от потребителски дефиниран тип (клас) се нарича обект. Класовете дават възможност да се моделират явления, които имат атрибути, представени като данни-елементи, и варианти на поведение, представени като функции-елементи. Понякога функциите-елементи в езиците за обектно-ориентирано програмиране се наричат методи, те се извикват в отговор на съобщения, изпратени към обекта.

Синтаксис на дефиниция на клас:

1.  class име\_на\_клас

2.  {

3.   тяло\_на\_клас

4.  };

Eтикетите public и private се наричат спецификатори за достъп до елементите. Данните-елементи и функциите-елементи декларирани след public до следващия спецификатор за достъп се наричат открити и те са достъпни при всяко обръщение от програмата към обект от класа. Данните-елементи и функциите-елементи декларирани след private до следващия спецификатор за достъп се наричат закрити и те са достъпни само за функциите-елементи от този клас. Спецификаторите за достъп завършват с ‘:’, могат да се появяват многократно в дефиницията и в произволен ред. Препоръчва се, обаче, в дефиницията на клас всеки спецификатор за достъп до елемент да се използва само веднъж, започвайки с public.

конструктор на този клас – това е специална функция-елемент, която инициализира данните-елементи на обект от този клас. Конструкторът на класа се извиква автоматично при създаване на обект от този клас. Обикновено класът има няколко конструктори - това се постига чрез предефиниране на функции.

Описване на функция извън класа – клас::функция

Функция-елемент, описана извън дефиницията на класа, може да бъде направена вградена чрез явно използване на ключовата дума inline

Функция със същото име, както на класа, но започваща с ‘~’ се нарича деструктор на този клас. Деструкторът изпълнява завършващи служебни действия на всеки обект на класа, преди паметта, разпределена за този обект да бъде повторно използвана от системата.

2.    Методи и параметри. Даннови членове. Инстанция на обект.

3. Класове и обекти. Преход от структура към клас.

Подобно на това, както същност от тип int се нарича променлива, то същност от потребителски дефиниран тип (клас) се нарича обект. Класовете дават възможност да се моделират явления, които имат атрибути, представени като данни-елементи, и варианти на поведение, представени като функции-елементи.

**Обектите представляват единство на своите физически и функционални характеристики**

**Множество от обекти със сходни характеристики на поведение и състояние се нарича клас.**

**Екземплярът е конкретен обект, принадлежащ на даден клас.**

**обединяват се  в едно цяло данни от *различен тип* и това е нужно, когато тези данни логически обособяват някаква сборна характеристика. В C++, обектите заместват структурите в C, но въпреки това има ситуации, в които те са полезни: структури се използват когато групираме няколко променливи, но те не се използват толкова интензивно в програмата, че да бъдат специфицирани като клас.**

**декларацията на структурата прилича много на декларацията на един клас, но въпреки това структурите и класовете се използват за съвсем различни цели: в класовете имаме функции, които оперират с неговите данни, докато обикновено в структурите нямаме такива функции.**

**В структурата може да има функции, както и функциите на класа може да бъдат премахнати. Структурата може да има и конструктор и деструктор.**

**Тук ще споменем няколко разлики: в структурата всичките и членове по подразбиране са публични (*public*), докато в класовете те са частни (*private*). Това означава, че в един клас може да се изпусне думата private. Също така в една структура може част от членовете да са private, но тогава ще са необходими функции на структурата за тяхната промяна и показване.**

**Обект (object) е близък до класическата структура (record), но съдържа освен**

**даннови полета собствени методи (процедури и функции) за работа с данните.**

**Класът е нещо много близко до обекта.**

**Обаче има две съществени разлики.**

**Първа разлика: класът може да съдъръжа управляем достъп до някои полета.**

**такива полета вече се наричат свойства (properties), те НЕ кореспондират пряко с променливи.**

**Втора разлика: класът е непременно динамична променлива, тоест компилатора**

**не може да заеме място за нея, това става в Run-Time, при споменаване на Create - конструктора, докато обектът може да е и статичен**

4. Конструктори и деструктори

След създаване на един обект, неговите елементи могат да бъдат инициализирани с помощта на функция конструктор. Конструкторът е функция-елемент, която има същото име, както името на класа. Конструкторът се извиква автоматично при създаването на обекта. За конструкторите не може да се задава тип на връщания резултат и те не връщат резултат.

Конструкторите могат да се предефинират и така да се определя множество от начални стойности за обектите на класа. Когато се дефинира обект от даден клас, между името на обекта и ‘;’ може в скобки да се зададе списък за инициализация на елементи – тези начални стойности се предават като параметри в конструктора на класа. Конструктор при който всички аргументи са по премълчаване или който няма аргументи, се нарича конструктор по премълчаване (по подразбиране). За всеки клас може да съществува само един такъв.

Ако за класа не е дефиниран конструктор, то компилаторът създава конструктор по премълчаване. Този конструктор не задава начални стойности, така че след създаването на обекта, данните в него могат да са некоректни. В конструктора може да се задават и стойности по подразбиране, такива че ако не бъде зададена стойност на променлива от класа да бъде дадена автоматично стойност.

Деструкторът е специална функция елемент на класа. Името на деструктора съвпада с името на класа, но пред него се поставя символ ‘~’. Мнемониката е, че ‘~’ означава поразрядно логическо отрицание. И така деструкторът може да се разглежда като отрицание на конструктора. Деструкторът на класа се извиква при унищожаване на обекта. Например, деструкторът за даден обект се извиква когато изпълнението на програмата напусне областта на действие на този обект. В действителност, самият деструктор не унищожава обекта, той извършва завършителни действия, преди системата да освободи паметта, в която е бил съхраняван обекта. Деструкторът няма параметри и не връща резултат. Не се разрешава предефиниране на деструктора, т.е. един клас може да има само един деструктор.На практика деструкторите рядко се използват в обикновени класове; деструкторите имат смисъл за класове, които използват динамично разпределение на паметта за обектите.

Конструкторите и деструкторите се извикват автоматично. Редът в който се изпълняват тези извиквания зависи от реда, в който изпълнението на програмата влиза или излиза от областите на действие на обектите. Ако деструктор на наследяващ клас бъде извикан то този на наследения няма да бъде потърсен автоматично и изпълнен, а обекта ще бъде унищожен с деструктора на наследника. Това може да се предотврати като в деструктора на наследника се викне този на родителя.

5. Типове променливи и оператори в клас. Основни типове данни.

6. Указатели и референции. Константи. Изброими типове.

В С++ има различни типове променливи – за цели числа, за числа с десетична запетая, за символи, низове и тн. Примери за типове променливи са : char, short int, int, float, double , bool и тн.

Оператори: В езика С++ има богат набор от оператори.В него са дадени също и средства за предефиниране.

Всеки оператор се харектеризира с :

-позиция на оператора спрямо аргомента му;

-приоритет

-асоциативност;

Позицията на оператора, спрямо аргумента му го определя като:

префиксен(оператора е пред единствения си аргумент --i),

инфиксен(оператора е между аргументите си- 4+2),

постфиксен(оператора е след аргументите си i++).

Приоритетът на оператора определя редът на изпалнение на операторите.Оператора с по-висок преоритет се изпалнявя преди оператора с по-нисак преоритет. (Пример: Преоритета на \* и / е по-висок от този на + и -).

Асоциативността определя реда на изпълнение на операторите с еднакъв преоритет. В С++ има ляво асоциативни и дясно асоциативни оператори. Ляво асоциативните оператори се изпалняват от ляво на дясно, а дясните - от дясно на ляво. В С++ не може да се дефинират нови оператори, но всеки съществуващ вече оператор с изключение на ::, . , ?: и .\* може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.

За предефинирането на оператор се съзадава операторна функция. Най-често

операторната функция е член-функция или приятелска функция на класа, за който е дефинирана.Съществуват две важни ограничения при предефинирането на оператор. Първо, приоритетът на операторите не може да бъде променян. Второ, броят на операндите, които приема този оператор, не може да бъде променян.

Операторите биват : за присвояване( = ), бинарни( + , - и тн), релационни и логически (примерно == и && ), унарни (пример: ++ ), оператор за индекс ( [ ] ) и др .

Основни типове данни

- Цели числа (int: 2-4 байта и лонг минимум 4 байта): 8, -54, 1289

- Дробни числа с плаваща – float или фиксирана точка - double: 0.01, 12.5, 73.0, 21.0E2

- Символни имена (идентификатори):

Аритметични операции в С++.

+/- - събиране и изваждане

\*,/,% - умножение, деление, остатък от деление

Особености:

- Приоритети - както в математиката (най-напред се извършват действия \*, / и % и след тях + и -)

- Използване на скоби за промяна на реда на изпълнение на аритметичните операции.

- Аргументна зависимост - действието на операцията зависи от типа на аргументите.

- Двата аргумента на аритметичните операции трябва да са от един и същи числов тип данни; резултатът е от същия тип.

- Разрешено е единият от аргументите да е от тип double/float, а другия - от тип int; тогава аргумента от тип int се преобразува (автоматично) на тип double и след това се извършва аритметичната операция - с аргументи тип double.

7. Предаване на данни като параметри. Accessor-методи. Mutator-метод.

програмистът се нуждае да изпрати допълнителна информация на функцията при нейното извикване. Това се извършва с така наречените [аргументи](http://kosnos.com/2008/issue17/cpp_terms_10-12.html#argument). Тези аргументи са параметри на които се задава стойност при извикване на самата функция.

Други функции пък връщат на извикващия определен тип данни, чрез променлива от зададеният тип.

Данни могат да бъдат предавани и чрез променлива тип указател, която показва пътя съм реалната стойност на параетър / позицията на данните в паметта.

В класа може да има 3 вида функции

\* функции които задават стойност на аргументите на класа

\*функции които връщат стойността на аргументите на класа

\*функции които извършват логическа операция с аргументи на класа по даден алгоритъм/свойство

Аccessor методите се използват за достъп до член данните на класа от външна за класа област. Самата функция принадлежи на класа, но е видима извън него, за да може да бъде достъпвана. accessor-а връща стойността на деден член на класа от същият тип, или оказва път към стойността в паметта. Функцията не приема аргументи, освен ако не са специфични за самият член на класа(примерно определен елемент от вече изграден масив на член на класа, а не целия член)

Mutator методите за използват за задаване/променяне на стойностите на членовете на класа. Организират външият достъп до член данните и са видими извън класа и неговите наследници. Не връщат резултат от даден тип (освен ако не връщат true/false за изпълнена промяна на стойности)

8. Общи понятия:функции; декларация на прототип; дефиниране на функция;

повикване на функция; работа с локални и глобални типове; дефиниране и

използване на overloaded функции.

Функция : **Функция на** [**C++**](http://bg.wikipedia.org/wiki/C%2B%2B) е начин да се разделят блоковете код на части. Те предоставят на програмиста начин да раздели програмата си така, че да не трябва непрекъснато да се пише един и същи код, а просто да извика функцията. Чрез функциите кода се поддържа малък, чист и функционален.

прототип: **Прототипът** указва на компилатора, че определена функция съществува, но тялото на функцията е някъде другаде. Бихте могли и да не пишете прототип, като сложите заглавната част и тялото над всички извиквания на функцията, но в някои случаи това е невъзможно. Освен това, прототипа помага за изчистването и разбираемостта на кода, което е и главната цел на функциите. Прототипът има следния синтаксис:

тип име (параметри);

дефиниране на функция: описва се логическият блок код на функцията

 тип име (параметри)

   {

тяло

   }

повикване на функция:

име (параметри); - ако функцията е без тип

променлива = име (параметри) - променливата е от същият тип като функцията

работа с локални и глобални променливи: локалните променливи не могат да бъдат използвани във функцията  затова те трябва да бъдат предадени като аргумент на тази функция (или да има указател към мястото им в паметта)

глобалните променливи могат да бъдат директно ползвани във функцията, стига да са декларирани над нея (над прототипа й).

дефиниране и използване на overloaded функции: Overload функции, т.е работят по натоварване. Такива фунцкии или процедури се наричат, overloaded, когато са декларирани фунции с едно и също име, с разлика вид и брой аргументи. При извикване на такива функции, зависи от това какви агрументи са подадени, за да се включи съответната функция.

9. Функции –методи. Глобални функции

10. Припокриване на функции

11. Локални променливи. Локални и/или глобални елементи на клас.

12. Взаимодействия на обекти.Понятие за абстракция и модулност.

13. Програмни практики: цифров часовник: диаграми на класове и обекти; проект и

програмни елементи.

14. Групиране на обекти.Колекции и итератори . Програмни практики: проект

‘notebook’; обектна структура, използване на колекции.

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

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

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

18. Проблеми в проектиране и ООП. Водещи практики.

19. Класове и обекти: разделяне на декларация и дефиниция. Създаване и

унищожаване на обекти.

20. Дефиниране на конструктори и деструктори. Методи на класа.

21. Дефиниране на връзки.

22. Наследяемост. Подтипове. Замествания. Полиморфизъм и променливи.

Наследяване в ООП наричаме възможността един клас, наричан **наследник**, да придобие свойства и действия на друг клас – **родител** (**базов** клас).

**Наследяемост:**

- Споделянето на атрибути и операции между класовете, базиращо се на йерархична

връзка

- Класовете могат да се дефинират по-общо и после да се конкретизират в по-точни

подкласове

- Всеки подклас включва или наследява (inherit) всички свойства на родителския си

клас (super class) и добавя свои уникални свойства

**Полиморфизъм** (polymorphism) е способността на един обект да съществува в

различни форми. Полиморфизмът дава възможност да използвате един и същ метод за

изпълнението на различни задания. В производните класове можете да променяте

реализацията на метод от базовия клас. Следователно, когато от един базов клас

създавате два производни класа, можете да създадете метод с едно и също име и в двата

класа. Но методът във всеки от производните класове изпълнява различни задачи.

Избира се кой метод да се извика в зависимост от това от кой обект се вика. Пример:

Animal \*ptr = **new** Cat(…);

ptr->speak();

ptr = **new** Dog(…);

ptr->speak();

23. Програмни практики: разнородни обекти в бази от данни. Проект: обектен модел,

класова диаграма, реализация. Подобрена версия с наследяемост. Суперклас и

подкласове. Наследяемост.

24. Създаване на проекти с дълбока йерархия.

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

26. Полиморфични променливи.

27. Наследяемост в ООП . Дефиниране на йерархията. Достъп до методи и данни на

различни нива.

28. Виртуални елементи и реализация на полиморфизма. Абстрактни класове и

абстрактни методи.

29. Програмни практики: наследяемост и поломорфизъм в програма.

30. Виртуални функции викане на виртуални функции на базов клас.

31. Вграждане на обекти. Copy – конструктори.

32. Присвоявания и обекти.

33. Референтни параметри (const или non-const). Работа с референции.

34. Връщане на референтни обръщения.

***Returning references***

• ***Дадена функция може да върне референция (това е еквивалентно на връщането на pointer)***

***const double& XY::GetConstX() const {return x};***

• ***Така декларирано функцията връща константна референция до XY object и може да се използва само от дясната страна на присвояването:***

***my.GetConstX() = 1.0; // is wrong!!!***

returning reference from a function

• ***Грешка в C е следното:***

***int \*GetInt()***

***{***

***int result = (int) (rand() / 1000);***

***return &result; // don’t do this!!***

***}***

• ***функцията връща pointer до стака,който ще се използва на др място след като функцията се изпълни (член-променливата липсва!!!) !!!***

• ***еквивалентна С++ грешка:***

***int& GetInt()***

***{***

***int result = (int)(rand() / 1000);***

***return result;***

***}***

***//the compiler is still returning a pointer to a temporary variable***

35. Конструиране на вградени обекти. Деструкция на вградени обекти.

Constructing embedded objects

II part orbiter

Planet  **Spaceship**

***1. Компилаторът има декларацията на обекта. От където следва че той знае колко памет трябва за Spaceship object и разпределя(allocates) тази памет.***

***2. всички вградени обекти (m\_current, m\_prior, m\_thrust) са конструирани***

***3. Orbiter constructor се извиква***

***4. m\_orrientation вградения обект се извиква.***

***5. Spaceship constructor функция се извиква.***

***Това е листа за конструкцията***

• ***The class design и синтаксиса на Spaceship constructor определят точно кой конструктур ще се извика(default, explicit or copy)***

**// This program demonstrates the order in which base and**

**// derived class constructors and destructors are called.**

**#include <iostream>**

**using namespace std;**

**// 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() // Constructor**

**{ cout << "This is the DerivedClass constructor.\n"; }**

**~DerivedClass() // Destructor**

**{ cout << "This is the DerivedClass destructor.\n"; }**

**};**

***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***// main function \****

***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***int main()***

***{***

***cout << "We will now define a DerivedClass object.\n";***

***DerivedClass object;***

***cout << "The program is now going to end.\n";***

***return 0;***

***}***

**Program output:**

***We will now define a DerivedClass object***

**This is the BaseClass constructor This is the DerivedClass constructor *The program is now going to end* This is the DerivedClass destructor This is the BaseClass destructor**

Destructing embedded objects

II part

**Нека Spaceship се унищожи:**

Той е производен на Orbiter class и има вградени обекти (като XY) дефиниран както и в базовия клас така и в производния клас:

1. spaceship destructor се извиква

2. m\_orientation вградения обект се унищожава

3. Orbiter destructor се извиква

4. m\_current, m\_prior and mass вградените обекти се унищожават

5. паметта за Spaceship е освободена

**remember mark:**

**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)**

**more about destruction**

•**Деструктурите не се наследяват. Компилаторът създава конструктор по подразбиране за всеки клас ако програмиста сам не създаде такъв. Този производен на класа деконструктор извиква базовия деструктор. Ако има липсващ код от производния деструктор, само деструкцията на базовите членове ще се извърши.**

**Деструкцията на производния клас е незавършена по този начин.**

•**Ако базовия конструктор е деклариран като виртуален:**

**virtual ~Orbiter() {}**

**Първо ще се унищожат всички елементи на SpaceShip и после ще извика**

**Orbiter destructor**

**Example:  *let’s try without virtual destructors*:**

***#include <iostream>***

***using namespace std;***

***// Animal is a base class. class Animal***

***{***

***public:***

***// Constructor***

***Animal()***

***{ cout << "Animal constructor executing.\n"; }***

***// Destructor***

***~Animal()***

***{ cout << "Animal destructor executing.\n"; }***

***};***

***// The Dog class is derived from Animal***

***class Dog : public Animal***

***{***

***public:***

***// Constructor***

***Dog() : Animal()***

***{ cout << "Dog constructor executing.\n"; }***

***// Destructor***

***~Dog()***

***{ cout << "Dog destructor executing.\n"; }***

***};***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***// main function \****

***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***int main()***

***{***

***// Create a Dog object, referenced by an***

***// Animal pointer.***

***Animal \*myAnimal = new Dog;***

***// Delete the dog object.***

***delete myAnimal;***

***return 0;***

***}***

***Program output:***

***Animal constructor executing Dog constructor executing Animal destructor executing***

To fix the previous problem: ***let’s try with virtual destructors*:**

***#include <iostream>***

***using namespace std;***

***// Animal is a base class. class Animal***

***{***

***public:***

***// Constructor***

***Animal()***

***{ cout << "Animal constructor executing.\n"; }***

***// Destructor***

***virtual ~Animal()***

***{ cout << "Animal destructor executing.\n"; }***

***};***

***// The Dog class is derived from Animal***

***class Dog : public Animal***

***{***

***public:***

***// Constructor***

***Dog() : Animal()***

***{ cout << "Dog constructor executing.\n"; }***

***// Destructor***

***~Dog()***

***{ cout << "Dog destructor executing.\n"; }***

***};***

***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***// main function \****

***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***int main()***

***{***

***// Create a Dog object, referenced by an***

***// Animal pointer.***

***Animal \*myAnimal = new Dog;***

***// Delete the dog object.***

***delete myAnimal;***

***return 0;***

***}***

Program output:

***Animal constructor executing Dog constructor executing Dog destructor executing Animal destructor executing***

36. Заделяне на обекти от динамичната памет. Проблеми, породени от

взаимодействията между обекти.

Създадените инстанции от даден клас съществуват докато не излязат извън обсега на

действие (scope, края на блока). За да запазиме инстанцията извън scope-а, може да я

създадем, като я заделим от динамичната памет където тя ще стои активна докато не я

освободим ръчно. Заделянето става чрез new, а изтриването чрез delete.

double \*buff = new double[1024];

Cat \*pCat = new Cat(…);

delete [] buff; delete pCat;

Инстанциите биват заделени в паметта, но за да ги достъпим трябва да използваме

указател сочещ към тях. Предимството е, че указателя ще изчезне когато излезе от

scope-а, но не и инстанцията. Достъпът до полета и методи до дадена инстанция чрез

указател става чрез стрелка ‘->’: pCat->speak(); Указателите ни позволяват да ги

насочваме към инстанции от клас който е наследен спрямо типа на указателя. Така

програмата става гъвкава и съвместима. Пример:

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. Възможно

е и конвертиране на класа от един тип към друг чрез cast-ване, но е опасно.

При работата със указатели, трябва да се внимава да не загубиме указател сочещ към

дадена инстанция, без да сме освободили инстанцията предварително, иначе се

получава memory leak и те никога в програмата няма да могат да бъдат освободени.

37. Виртуални деструктори.

Деструктурът е функция, която се извиква при унищожаването на даден обект. Името

на деструктура на един клас е името на класа, предшествано от ~. Локалните обекти се

унищожават, когато излязат извън областта си на видимост, а глобалните – когато

завърши изпълнението на програмата. Ако е заделен ръчно, той бива освободен при

изход на програмата или чрез delete.

При наследяването на класове дестурктурът на един производен клас трябва да се

изпълни преди деструктура на базовия клас. Ако деструкторът на базовия клас се

изпълняваше пръв, това би довело до унищожаването на производния клас.

Пример:

class Base {

public:

~Base(){ ... }

};

class Derived: public Base {

public:

~Derived(){ ... }

};

void main() {

Base \*Var = new Derived();

delete Var;

}

Създава се указател от типа на базовия клас, сочещ към нов обект от типа на

производния. При изтриване на указателя от типа на базовия клас директно ще се

извика деструктора на базовия клас. Когато се извика деструктора на базовия клас,

полетата липсващи в базовия клас Base специфични за наследения клас няма да бъдат

унищожени => разрушаването на производния клас ще остане незавършено и

специфичните полета няма да бъдат освободени от паметта т.е. програмата ще стане

следното:

Constructor: Base

Constructor: Derived

Destructor : Base

Проблемът се решава като се обяви деструкторът на базовия клас за виртуален.

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. Т.е. ако сложиме:

virtual ~Base(){ ... }

В този случай като изтрием указателя от базовия клас, компилаторът ще провери кой

подходящ конструктор да извика, а именно този на наследеният

38. Приятелски класове и приятелски функции.

Приятелската функция не е член-функция на класа, но тя има достъп до *private*

членовете на класа. Приятелски функции се използват, когато е необходимо една

функция да има достъп до *private* членовете на даден клас.

Декларацията на приятелската функция, в един клас, може да се намира в *public* или в

*private* и се характеризира с ключовата дума friend, която се поставя в началото на

декларацията на метода. Една приятелска функция се дефинира като обикновен метод,

но не е член на класът. Затова те не получават като първи параметър по неявен начин

this. Поради тази причина е необходимо подаването на обект по явен начин като

параметър на функцията.

Когато една член-функция прави обръщение към private елемент, компилатора знае че

този метод е от този клас и има достъп до това поле. Една приятелска функция обаче не

е свързана с обект. На нея просто и е предоставен достъп до private елементите на даден

клас.

Приятелска фуннкция не се наследява т.е когато един базов клас съдържа приятелска

функция, тя не е приятелска на производния клас.

Един клас за да бъде приятелски на друг клас трябжа да буде деклариран във

въпросният клас.Това се получава по подобен начин както приятелските функции, само

че пишем директно името на класа след friend.

39. Софтуерни контракти. Пред-условия, пост-условия и инварианти към методи на

клас. Синтаксис в .NET. 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 ново-генерираните 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> ....

40. 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 инстина 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.

41. 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 опище своите собствени инварианти.

 public abstract class DomainObject

 { public abstract Boolean IsValid();

 [Pure] private Boolean IsValidState()

 {

 return IsValid();

 }

 [ContractInvariantMethod] private void ObjectInvariant()

 {

 Contract.Invariant(IsValidState());

 }

42. Статични членове на клас.

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.

43. Припокриване на оператори. Същност, ограничения. Реализация.

**Предефиниране на операции:** В С++ всяка съществуваща унарна или бинарна

операция, в която участва поне един обект от някакъв клас, може да бъде

предефинирана от програмиста. Това дава възможност класовете да бъдат

интерпретирани, като нови типове данни, които могат да се използват по начин

аналогичен на базовите типове. Така например, ако бъде дефиниран клас вектор и v1 и

v2 са обекти от този клас, то е възможно използването на изрази като: v1+v2, v1-v2,

v1\*v2, v1/v2...

Предефинирането на операции се осъществява чрез операторни функции.

Операторната функция е член-функция или приятелска функция на класа, за който е

дефинирана.

Предефиниране чрез член-функции.

Общата форма на една член-функция оператор е следната:

<тип-резултат> <име-на-клас>::<operator>#(<списък аргументи>)

{ ... }

Типът на връщания резултат най-често е същият като типа на класа, за който е

дефиниран операторът. Операторът, който се предефинира, замества # в конструкцията.

Предефиниране на бинарни оператори.

Когато една операторна член-функция предефинира бинарен оператор, функцията ще

приема само един параметър. Този параметър ще получава обекта, който е от дясната

страна на оператора. Обектът от лявата страна е този, който генерира обръщението към

операторната функция и точно той се предавa неявно на функцията посредством

указателя this.

Предефиниране на унарни оператори.

Когато се предефинира даден унарен оператор посредством операторна член-функция,

функцията не приема параметри. Тъй като операндът е само един, то той е този, който

генерира обръщението към операторната функция.

Предефиниране чрез приятелски функции.

Тъй като една приятелска функция не получава this указател, в случай на бинарен

оператор това означава, че се предават директно и двата операнда. При унарните

оператори се предава единствен операнд.

Не може да се използва приятелска функция, за да се предефинира оператора за

присвояване. Оператора за присвояване може да бъде предефиниран само от

операторна член-функция.

Всички традиционни оператори могат да бъдат припокривани, включително и new,

delete и конвертиращите оператори. Не е възможно да бъдат предефенирани ., .\*, ::, ?:

операторите.

Правила за припокриване:

- Не може да имаме нов оператор

- Не може да се променя броя на операндите, взимани от оператора.

- Не може да се променят приоритетите или асоциавността на операторите

44. Програмни практики: припокриване на аритметични операции.

Припокриващите се оператори са имплементирани като функции.Дефинират се като

‘operatorx’, където ‘x’ е операторът.

Например за припокриване с оператор за сумиране, можете да дефинирате функцията

като operator+.

Подобно е положението и при изваждане и равенство “-“ и “=”.

Операторите, използвани в с++ работят само с примитивни типове, не със наши

собствено създадени, тъй като компилатора незнае как да използва операторите за

нашите типове.

Всички традиционни оператори могат да бъдат припокривани, включително и new,

delete и конвертиращите оператори. Не е възможно да бъдат предефенирани ., .\*, ::, ?:

операторите.

Правила за припокриване:

- Не може да имаме нов оператор

- Не може да се променя броя на операндите, взимани от оператора.

- Не може да се променят приоритетите или асоциавността на операторите

За пример \* винаги има по-висок приоритет от +.

struct Complex {

....

Complex operator+( Complex &other );

};

Complex Complex::operator+( Complex &other )

{ return Complex( re + other.re, im + other.im ); }

45. Преобразувания и операции -преобразувания.

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.

46. Пример - Microsoft Foundation Class Library (MFC).

Microsoft Foundation Class Library (MFC Library)

The Microsoft Foundation Class (MFC) Library is a collection of class es (generalized definitions used in object-oriented programming ) that can be used in building application program s. The classes in the MFC Library are written in the C++ programming language. The MFC Library saves a programmer time by providing code that has already been written. It also provides an overall framework for developing the application program.

There are MFC Library classes for all graphical user interface elements (windows, frames, menus, tool bars, status bars, and so forth), for building interfaces to database s, for handling events such as messages from other applications, for handling keyboard and mouse input, and for creating ActiveX control s.

47. Обектен дизайн : принципи на SOLID, open/closed принцип, принцип на

регламентираната отговорност.

Solid е популярен акроним, съдържащ инициалите на 5 принципа на софтуерния дизайн: Single responsibility(Регламентирана Отговорност), Open/Closed , Interface segregation , Dependency inversion и Liskov substitution principle.

Open/closed принцип:

обектите(методи/функции/класове) Трябва да са отворени за разширяване, но затворени за модифициране.

Single Responsibility Principle:

Когато трябва да напишем клас , които да не променяме е да го напишем,така че да изпълнява само прави само 1 нещо.Принципът е не само да пишем малки класове и методи - всеки клас трябва да съдържа определен набор от свързани функции.

48. Обектен дизайн : принцип на верижната отговорност,

Принципът е , че според нуждите можем да добавяме различни класове и методи към интерфейса. Този модел ни позволява чрез минимален риск да добавяме реализации, вместо единичен клас да опише всички реализации.

 public class OrderProcessingModule

 {

 private IOrderHandler[] \_handlers;

 public OrderProcessingModule()

 {

 \_handlers = new IOrderHandler[]

 {

 new InternationalOrderHandler(),

 new SmallDomesticOrderHandler(),

 new LargeDomesticOrderHandler(),

 };

 }

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

Двойния dispatch ( пренасочване ) ни позволява да превърнем всички варианти в подкласове, като по този начин не нарушаваме съществуващите интерфейстни връзки.

 public interface IPresenter { void SetupView(IApplicationShell shell);}

 public class ApplicationController {

 private IApplicationShell \_shell;

 public ApplicationController(IApplicationShell shell) { \_shell = shell; }

 public void ActivateScreen(IPresenter presenter) {

 // Set up the new screen using Double Dispatch

 presenter.SetupView(\_shell); // redirect to direct connection

 }

 }

С общи думи : обектът и параметъра се подават когато се задейства функцията или програмата.

За да се посочи точния обект по време на изпълнението, се използва Visitor Pattern. Toвас е нарича Двойна dispatch ( пренасочване ) форма. обекта трябва да съдържа метод Visit(), които да е свързан с обект dispatcher. Oбектът Dispatcher има набор от методи, които му позволяват да се насочи към конкретен тип от обекти.

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

Функции които ползват указатели/референции към базов клас, трябва да могат

да ползват обекти от наследници 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.

51. Нови стандартизирани конструкции в C++. Асинхронни задачи, нишки и локални

за тях данни.

52. Конструктори в C++ за синхронизация: атомични типове и условни променливи,

мютекси и заключвания.

53. 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 аргуманта;

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.

54. Генетични типове и наследяемост. Синтактично подменяне на генетичен тип.

Обработка на генетични типове. Ограничители.

55. 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), Koeто е загуба на ресурс и бързо действие

Ето един стандартен в FCL интерфейс:

 public interface IEnumerator<T> : IDisposable, IEnumerator {

 T Current { get;}

 }

Eто 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;

 }

55. Съвременни езикови конструктори: lambda expression; функции-обекти.

Сравнение, анализ.

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

В С обектите в паметта се разполагат с командите:

 calloc()

 malloc()

 realloc()

Паметта се освобождава с функцията free()

В С++ за разполагане на обекти в паметта може да се използва и оператора new, а за освобождаване на паметта се използва оператора delete. В С++ може да се използват и командите от С.

malloc(size\_t size);

 \*Локализира битовете и връща указател към локализираната памет.

 \* Паметта не е изчистена.

Free(void \* p)

 \*освобождава паметта към която сочи p

 \*ако командата free(p) вече е била извикана може да се появи неопределено поведение.

 \* Ако р е NULL никаква операция не се извършва.

realloc (void \*p, size\_t size);

 \*Променя големината на блока с памет към който сочи р към размерни битове.

 \* Ново локализираната памет е неинициализирана.

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

57. Управление на памет в конзолен режим и в Linux системи: служебни структури в паметта. Начин на работа на системата за освобождаване (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

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}.

58. Препълване на буфер. Поглед отвътре. Използване за недобросъвестно вмъкване на код. Пример.

При динамично разполагане на памет може да се появи препалване на буфера, информация бива записвана извън границите на буфера което предоставя възможност на недоброжелател да вмъкне код с който да промени поведението на програмата. Това може да се осъществи с техниките Unlink и Frontlink.

Пример за Unlink macro:

#define unlink(p, BK, FD) {

FD = P->fd;

BK = P->bk;

FD-> bk =BK;

BK->fd = FD;

}

Пример за код за Buffer Overflow:

intmain(intargc, char \*argv[]) {

char \*first, \*second, \*third;

first = malloc(666);

second = malloc(12);

third = malloc(12);

strcpy(first, argv[1]);

free(first);

free(second);

free(third);

return(0);

}

Командата strcpy() е без граници което допуска препълване на буфера. free(first); освобождава първия блок памет.free(second); ако паметта е свободна, я слепва към първия блок, за да провери дали паметта е свободна командата free() проверява PREV\_INUSE бит на третия блок.

Недоброжелателят може да запише информация в адресното пространство на 2рия блок примерно -4, по този начин третия блок изглежда да е 4 байта преди 2рия блок. free() операцията се извършва и се слепва 2рия блок към първия блок, така вече пойтерите на първия блок слочат към недоброжелателен код.



59. 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.

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

Опасностите при двойно освобождаванена паметта се появява при опит блок памет да бъде освободен 2 пъти без да бъде локализиран между освобождаванията.

За да се злоупотреби с двойното освобождаване на паметта трябва да бъдат изпълнени 2 условия:

 \*Блока с памет трябва да е изолиран в паметта

 \*Мястото кадето ще бъде поставен блока трябва да бъде празно





61. 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>,МЕМ\_Reserve,<права за достъп>);

Заделянето става по страници(например 4К) и по необходимост, в ОП и swap file от резервираната, след което може да се използва паметта. След изчерпването й се генерира exeption: exeption\_guard\_page.





62. Служебни структури в динамичния менижмънт на памет в Windows.





63. 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 пойнтерите 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-a ни трябва адрес който да е изпълним, откриването на този адресс е трудно, но възможно. Друг начин е да се получи достъп до адрес ексептшън, който да бъде заменен.

64. Съпоставяне на файл с Оперативна Памет. Съвети при управление на паметта.



