**60.Делегати.Дефиниране, използване**

**Делегатите** са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура и връщан тип.

### Статични или екземплярни методи

Делегатите могат да сочат както към методи на инстанция на класа, в който са декларирани, така и към статични методи. Това представлява удобство, защото можем да използваме делегат, без да сме създали инстанция на съдържащия го клас. Така се спестява създаването на допълнителна инстанция на клас. Друга възможност е да се отложи създаването на инстанция на делегат докато тя стане необходима. За целта можем да дефинираме свойство на класа, който ползва делегата, и в get метода на свойството да създадем делегата.

### Пример за делегат

Следващият пример демонстрира деклариране на делегат, инстанциране на делегат и извикване на метод, сочен от него.

// Declaration of a delegate

public delegate void SimpleDelegate(string aParam);

class TestDelegate

{

public static void TestFunction(string aParam)

{

Console.WriteLine("I was called by a delegate.");

Console.WriteLine("I got parameter {0}.", aParam);

}

public static void Main()

{

// Instantiation of а delegate

SimpleDelegate simpleDelegate =

new SimpleDelegate(TestFunction);

// Invocation of the method, pointed by a delegate

simpleDelegate("test");

}}

### Единични (singlecast) делегати

**Единичните делегати** наследяват класа System.Delegate. Тези делегати извикват точно един метод. В списъка си на извикване имат единствен елемент, съдържащ референция към метод.

### Множествени (multicast) делегати

**Множествените делегати** наследяват класа System.MulticastDelegate, който от своя страна е наследник на класа на System.Delegate. Те могат да викат един или повече метода. Техните списъци на извикване съдържат множество елементи, всеки рефериращ метод. В тях може един и същ метод да се среща повече от веднъж. При извикване делегатът активира всички реферирани методи. Множествените делегати могат да участват в комбиниращи операции.

**54.Събития.Кратък пример**

**Събитията** могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програмиране компонентите изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел е много характерен например за графичните потребителски интерфейси, където контролите уведомяват чрез събития други класове от програмата за действия от страна на потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натиснат. Разбира се, събития могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която като част от функционалността влиза трансфер на файлове. Приключването на трансфера на файл може да се съобщава чрез събитие.

### Събития – пример

В настоящия пример се разглежда дефинирането и използването на събития като се спазва утвърдената конвенция в .NET Framework. Демонстрира се изпращане и получаване на събития.

// A delegate type for hooking up change notifications

public delegate void TimeChangedEventHandler(

object aSender, TimeChangedEventArgs aEventArgs);

// A class that inherits System.EventArgs and adds

// information for the current time

public class TimeChangedEventArgs : EventArgs

{

private int mTicksLeft;

public TimeChangedEventArgs(int aTicksLeft)

{

mTicksLeft = aTicksLeft;

}

public int TicksLeft

{

get

{

return mTicksLeft;

}

}

}

public class Timer

{

private int mTickCount;

private int mInterval;

// The event that will be raised when the time changes

public event TimeChangedEventHandler TimeChanged;

public Timer(int aTickCount, int aInterval)

{

mTickCount = aTickCount;

mInterval = aInterval;

}

public int TickCount

{

get

{

return mTickCount;

}

}

public int Interval

{

get

{

return mInterval;

}

}

// The method that invokes the event

protected void OnTimeChanged(int aTick)

{

if (TimeChanged != null)

{

TimeChangedEventArgs args =

new TimeChangedEventArgs(aTick);

TimeChanged(this, args);

}

}

public void Run()

{

int tick = mTickCount;

while (tick > 0)

{

System.Threading.Thread.Sleep(mInterval);

tick--;

OnTimeChanged(tick);

}

}

}

public class TimerDemo

{

// The event handler method

private static void Timer\_TimeChanged(object aSender,

TimeChangedEventArgs aEventArgs)

{

Console.WriteLine("Timer! Ticks left = {0}",

aEventArgs.TicksLeft);

}

public static void Main()

{

Timer timer = new Timer(10, 1000);

timer.TimeChanged +=

new TimeChangedEventHandler(Timer\_TimeChanged);

Console.WriteLine(

"Timer started for 10 ticks at interval 1000 ms.");

timer.Run();

}

}

**56.Пакетирани типове. Проблеми с достъпа.**

## Опаковане (boxing) и разопаковане (unboxing) на стойностни типове

Вече обяснихме, че стойностните типове се съхраняват в стека на приложението и не могат да приемат стойност null, докато референтните типове съдържат указател (референция) към стойност в динамичната памет и могат да бъдат null.

Понякога се налага на референтен тип да се присвои обект от стойностен тип. Например може да се наложи в System.Object инстанция да се запише System.Int32 стойност. CLR позволява това благодарение на т. нар. "**опаковане**" на стойностните типове (**boxing**).

В .NET Framework стойностните типове могат да се използват без преобразуване навсякъде, където се изискват референтни типове. При нужда CLR опакова и разопакова стойностните типове автоматично. Това спестява дефинирането на обвиващи (wrapper) класове за примитивните типове, структурите и изброените типове, но разбира се, може да доведе и до някои проблеми, които ще дискутираме по-късно.

### Опаковане (boxing) на стойностни типове

Опаковането (boxing) е действие, което преобразува стойностен тип в референция към опакована стойност. То се извършва, когато е необходимо да се преобразува стойностен тип към референтен тип, например при преобразуване на Int32 към Object:

|  |
| --- |
| int i = 5;  object obj = i; // i се опакова |

Всяка инстанция на стойностен тип може да бъде опакована чрез просто преобразуване до System.Object. Ако един тип е вече опакован, той не може да бъде опакован втори път и при преобразуване към System.Object си остава опакован само веднъж.

CLR извършва опаковането по следния начин:

1. Заделя динамична памет за създаване на копие на обекта от стойностния тип.
2. Копира съдържанието на стойностната променливата от стека в заделената динамична памет.
3. Връща референция към създадения обект в динамичната памет.

При опаковането в динамичната памет се записва информация, че референцията съдържа опакован обект и се запазва името на оригиналния стойностен тип.

**52.Стойностни типове.**

#### Стойностните типове и паметта

Стойностните типове заемат необходимата им памет в стека в момента на декларирането им и я освобождават в момента на излизане от обхват (при достигане на края на програмния блок, в който са декларирани). Заделянето и освобождаване на памет за стойностен тип реално се извършва чрез единично преместване на указателя на стека и следователно става много бързо.

Горното обяснение е малко опростено. Всъщност ако стойностен тип има за член-данни само стойностни типове, при инстанциране целият тип ще се задели в стека. Ако, обаче, стойностен тип (например структура) съдържа като член-данни референтни типове, стойностите им ще се запишат в динамичната памет.

#### Предаване на стойностни типове

При извикване на метод стойностните типове се подават по стойност, т.е. предава се копие от тях. При подготовка на извикването на метод CLR копира подаваните като параметри стойностни типове от оригиналното им местоположение в стека на ново място в стека и подава на извиквания метод направените копия. Ако извикваният метод промени стойността на подадения му по стойност параметър, при връщане от извикването промяната се губи. Това поведение важи, разбира се, само ако параметрите се подават по подразбиране, без да се използват ключовите думи в C# ref и out, които ще разгледаме по-нататък в следващите теми.

**26.Изключения в .NET. Дефиниране на собствени изключения.**

Изключенията в .NET са класическа имплементация на изключенията от ООП, макар че притежават и допълнителни възможности, произтичащи най-вече от предимствата на управлявания код.

В .NET Framework управлението на грешките се осъществява предимно чрез изключения. Всички операции от стандартната библиотека на .NET (Framework Class Library) сигнализират за грешки посредством **хвърляне** (**throw**, **raise**) на изключение. .NET програмистите трябва да се съобразяват с изключенията, които биха могли да възникнат и да предвидят код за тяхната обработка в някой от извикващите методи.

Изключение може да възникне поради грешка в нашия код или в код който извикваме (примерно библиотечни функции), при изчерпване на ресурс на операционната система, при неочаквано поведение в .NET средата (примерно невъзможност за верификация на даден код) и в много други ситуации.

В повечето случаи едно приложение е възможно да се върне към нормалната си работа след обработка на възникнало изключение, но има и ситуации в които това е невъзможно. Такъв е случаят при възникване на някои **runtime** изключения. Пример за подобна изключителна ситуация е, когато една програма изчерпа наличната работна памет. Тогава CLR хвърля изключение, което сигнализира за настъпилия проблем, но програмата не може да продължи нормалната си работа и единствено може да запише състоянието на данните, с които работи (за да минимизира загубите), и след това да прекрати изпълнението си.

Всички изключения в .NET Framework са обекти, наследници на класа System.Exception, който ще разгледаме в детайли след малко. Всъщност, съществуват и изключения, които не отговарят на това изискване, но те са нестандартни и възникват рядко. Тези изключения не са съвместими със CLS (Common Language Specification) и не могат да се предизвикат от .NET езиците (C#, VB.NET и т. н.), но могат да възникнат при изпълнение на неуправляван код.

Изключенията носят в себе си информация за настъпилите грешки или необичайни ситуации. Тази информация може да се извлича от тях и е много полезна за идентифицирането на настъпилия проблем. В .NET Framework изключенията пазят в себе си името на класа и метода, в който е възникнал проблемът, а ако асемблито е компилирано с дебъг информация, изключенията пазят и името на файла и номера на реда от сорс кода, където е възникнал проблемът.

Когато възникне изключение, изпълнението на програмата спира. CLR средата запазва състоянието на стека и търси блока от кода, отговорен за прихващане и обработка на възникналото изключение. Ако не го намери в границите на текущия метод, го търси в извикващия го метод. Ако и в него не го намери, го търси в неговия извикващ и т. н. Ако никой от извикващите методи не прихване изключението, то се прихваща от CLR, който показва на потребителя информация за възникналия проблем.

Изключенията улесняват писането и поддръжката на надежден програмен код, като дават възможност за обработката на проблемните ситуации на много нива. В .NET Framework се позволява хвърляне и прихващане на изключения дори извън границите на текущия процес.

## Собствени изключения

В .NET Framework програмистите могат да дефинират собствени класове за изключения и да създават класови йерархии с тях. Това осигурява много голяма гъвкавост при управлението на грешки и необичайни ситуации. В по-големите приложения изключенията се разделят в логически в категории и за всяка категория се дефинира по един базов клас, а за конкретните представители на категориите се дефинира по един клас-наследник.

### Дефиниране на собствени изключения

За дефинирането на собствени изключения се наследява класът System. ApplicationException и му се създават подходящи конструктори и евентуално му се добавят и допълнителни свойства, даващи специфична информация за проблема. Препоръчва се винаги да се дефинират поне следните два конструктора:

|  |
| --- |
| MyException(string message);  MyException(string message, Exception InnerException); |

Въпреки, че не е задължително, силно се препоръчва имената на изключенията да завършват на "Exception", например OrderException, CustomerNotFoundException, InvalidCredentialsException и т. н.

Веднъж дефинирани, собствените класове за изключения могат да се ползват по същия начин, както и системните изключения.

**27.Правила за работа с изключения в .NET среда.**

Изключенията са много мощен механизъм за обработка на грешки, но ако се използват неправилно, могат да доведат до много трудни за откриване проблеми. Затова ще посочим някои препоръчвани практики при работата с изключения:

* catch блоковете трябва да са подредени така, че да започват от изключенията най-ниско в йерархията и да продължават с по-общите. Така ще бъдат обработени първо по-специфичните изключения и след това по общите. В противен случай кодът за по-специфичните никога няма да се изпълни.
* Всеки catch блок трябва да прихваща само изключенията, които очаква (и знае как да обработва), а не всички. Лоша практика е да се прихващат всички изключения тъй като различните видове изключения изискват различна обработка и специфични действия за справяне с възникналата проблемна ситуация.Избягвайте конструкциите catch (Exception) {…} или просто catch {…}.
* При дефиниране на собствени изключения трябва да се наследява System.ApplicationException, а не директно System.Exception. По този начин може да се направи разграничение на това дали изключението е от .NET Framework или е от приложението.
* Имената на класовете на всички изключения трябва завършват на Exception, например OrderException, InvalidAccountException и т. н. Това прави кода по-разбираем и по-лесен за поддръжка.
* При създаване на инстанция на изключение винаги трябва да й се подава в конструктора подходящо съобщение. Това съобщение ще бъде достъпно по-късно чрез свойството Message на изключението и ще помогне на програмиста, който използва дадения клас, по-лесно да идентифицира проблема.
* Изключенията могат да намалят значително производителността на приложението, понеже всяко хвърлено изключение инстанцира клас (това отнема време), инициализира членовете му (това също отнема време), извършва търсене в стека за подходящ catch блок (и това отнема време) и накрая след като инстанцията стане неизползваема, тя се унищожава от garbage collector (и това също отнема време). Затова, когато е възможно се препоръчва да се прави проверка дали е възможно дадено действие, а не да се разчита на обработката на възникналото изключение. Прекомерното използване на изключенията се отразява на производителността.
* Някои изключения могат да възникват по всяко време без да ги очакваме (например System.OutOfMemoryException). Добра практика е да се централизира прихващането на този тип изключения на най-високо ниво например в Main() метода на програма и да се направи елегантно прекратяване на изпълнението на програмата.
* Изключенията трябва да бъдат хвърляни само при ситуации, които наистина са изключителни и трябва да се обработят. В нормалния ход на програмата (когато не възникват проблеми) не трябва да се хвърлят изключения.

**2.Въведение в .NET среда. .NET framework архитектура. Поддържане на единна езикова среда.**

.NET Framework e среда за разработка и изпълнение на приложения за .NET платформата. Тя предоставя програмен модел, библиотеки от типове и единна инфраструктура за разработка на приложения и поддържа различни езици за програмиране.

Приложенията, базирани на .NET Framework, се компилират до междинен код (на езика IL) и се изпълняват контролирано от средата за изпълнение на .NET Framework. Компилираният .NET код се нарича още управляван код и може да работи без да се прекомпилира върху различни платформи, за които има имплементация за .NET Framework (Windows, Linux, FreeBSD).

Можем да разделим .NET Framework на два основни компонента:

* **Common Language Runtime (CLR)** – средата, в която се изпълнява управляваният код на .NET приложенията. Представлява виртуална машина, която контролирано изпълнява .NET кода и осигурява различни услуги, като управление на сигурността, управление на паметта и др.
* **Framework Class Library** **(FCL)** – представлява основната библиотека от типове, които се използват при изграждането на .NET приложения. Съдържа основната функционалност за разработка, необходима за повечето приложения, като вход/изход, връзка с бази данни, работа с XML, изграждане на уеб приложения, използване на уеб услуги, изграждане на графичен потребителски интерфейс и др. Стандартните класове и типове от FCL можем да използваме навсякъде, където има инсталиран .NET Framework.

## Архитектура на .NET Framework

Архитектурата на .NET Framework често пъти се разглежда на нива, както това е направено на следната схема:

A description...

**57.Референтни типове**

### Референтни типове (reference types)

Типовете по референция (референтни типове) са класовете, интерфейсите, масивите и делегатите. Например:

|  |
| --- |
| class Foo: Bar, IFoo {...} // клас  interface IFoo: IBar {...} // интерфейс  string[] a = new string[5]; // масив  delegate void Empty(); // делегат |

На всички типове в C# съответстват типове от общата система от типове (Common Type System – CTS) на .NET Framework. Например, на примитивния C# тип int съответства типа System.Int32 от CTS.

**33.Работа с динамични блокове памет.Предимства на използването на собствени динамични блокове.**

Има 2 функции за работа с хипове:

- С++ функции MFC- по – универсални

- с API функции на ОС – специфични и когато се гони бързо действие и големи обеми.

Има 2 вида хип:

- автоматично зареждане на приложението(по подразбиране)

- динамично зареждане – формиране на собствен потребителски хип.

Някои препоръки за работа със собствения хип:

\* в рамките на своя клас се създава собствен хип, така че методите на класа да работят със собствения хип. Предимствата са:

- безопасност – другата нишка да се намеси в хипа.

- до голяма степен се избягва фрагментацията.

\*в С++ може да се предефинира New и Delete - чрез предефинирането на New може да се направи оптимално заделяне на памет. Схема реализираща предефинирането на New е следната:

1) проверява дали е заделен локален хип с New ако не е го заделяме.

2) заедно със създаването се задава и брояч на New и Delete.

3) заделя се необходимия брой байтове, които се заделени

4) инкрементираме брояча

Схема реализираща предефинирането на Delete е следната:

1) освобождава блока

2) декремнтира брояча

3) проверява ако е 0 унищожава handler към файла

Има функция – heapwin за създаден вече (от нас) хип. Големите блокове алокират собствен хип и връщат указател към тях. За малки блокове използват памет от вече алокиран хип.

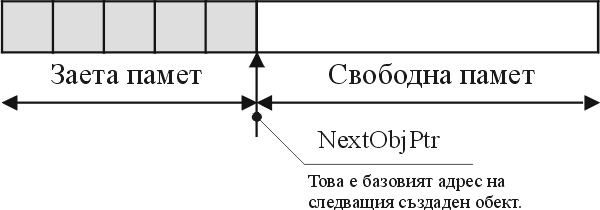
За големи блокове след интензивно използване на Delete трябва ни чести интервали за викане – heapwin, който всъшност освобождава тези блокове от хипа.

Когато CLR се инициализира, той заделя регион от последователни адреси в паметта. Това е т.нар. **динамична памет** или **managed heap**.

За разлика от стойностните типове, чиято памет се заделя в стека и се освобождава веднага, след като променливата излезе от обхват, паметта, нужна за референтните типове, винаги се заделя в managed heap.

В тази секция ще разгледаме как се осъществява заделянето на памет в хийпа.

В .NET, динамичната памет винаги се запълва последователно отляво надясно. Можете нагледно да си представите управлявания хийп като конвейер, при който обектите се добавят един след друг върху лентата (паметта), като всеки следващ е плътно долепен до предишния. За да е възможно това, хийпът поддържа указател, т.нар. NextObjPtr, който сочи адреса на който ще се добави следващият създаден обект. Фигурата илюстрира това описание:



Когато процесът се стартира, динамичната памет не съдържа никакви обекти и NextObjPtr е установен да сочи към базовия адрес от хийпа.



За да създадем обект в managed heap, използваме код, подобен на този:

|  |
| --- |
| SomeObject x = new SomeObject(); |

C# компилаторът превежда кода в IL newobj инструкция:

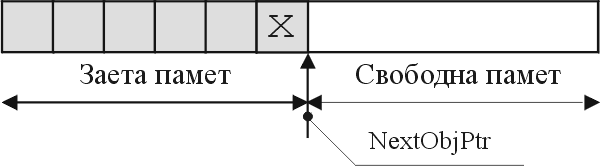
|  |
| --- |
| newobj instance void MyNamespace.SomeObject::.ctor() |

Когато тази инструкция се изпълнява, CLR действа по следния начин:

* Изчислява размера, необходим за полетата на новия обект и всичките му родителски обекти.
* Към получения размер прибавя размера на MethodTablePointer и SyncBlockIndex (специални служебни полета). При 32-битовите системи, тези две полета добавят 8 байта към размера на всеки обект, а при 64-битовите системи – 16 байта.
* Прибавя получената стойност към указателя NextObjPtr. Ако в managed heap има достатъчно място, паметта се заделя, извиква се конструкторът на обекта, който я инициализира, и адресът на обекта се връща от new оператора. Ако CLR установи, че мястото в паметта е недостатъчно, се стартира garbage collector. След като той приключи работа, CLR опитва отново да създаде обекта. Ако и тогава няма достатъчно памет, хийпът се увеличава, а ако това е невъзможно, new операторът предизвиква OutOfMemoryException.

Значението на полетата MethodTablePointer и SyncBlockIndex, които CLR създава за всеки обект от управлявания хийп, е извън темата на тази глава. Накратко, MethodTablePointer, както показва името му, съдържа указател към адреса на таблицата с методите на дадения тип, а SyncBlockIndex се използва при синхронизацията на обекта между нишките. За целите на настоящото изложение, просто трябва да запомните, че всеки един обект от хийпа съдържа тези две полета, които увеличават размера му с 8 или 16 байта, съответно при 32 и 64 битовите системи.

След като обектът е успешно създаден, CLR установява NextObjPtr на първия свободен адрес, непосредствено след края на новия обект, както е показано на следващата фигура.



Вероятно се досещате, че този начин за заделяне на памет в managed heap работи много бързо, защото физически се имплементира с прибавянето на стойност (размерът на обекта) към указателя NextObjPtr. Всъщност скоростта на създаване на референтен тип в managed heap е съпоставима със заделянето на памет в стека. За разлика от .NET, в C++ runtime heap заделянето на памет е значително по-тежка операция, при която след изчисляването на размера на обекта първо се търси достатъчно голям блок свободна памет и едва след това обектът може да бъде създаден.

Освен това, тъй като паметта се запълва последователно, когато създаваме обекти един след друг, те физически ще се намират на близки адреси в паметта. Това може значително да подобри производителността в някои ситуации, тъй като обектите, създадени приблизително по едно и също време обикновено са логически свързани и приложението често ги използва заедно (представете си например локални променливи в тялото на даден метод). Така е възможно всички обекти, които дадена част от кода използва, да се намират в кеша на процесора и работата с тях ще е много бърза.

Трябва да се има предвид, обаче, че освобождаването на памет от хийпа е сложна и времеотнемаща операция. Тя се извършва от системата за почистване на паметта, когато има недостиг на памет. Почистването на паметта и алгоритъмът, по който то се извършва, ще разгледаме подробно в следващите секции.

**36.Финализация в .NET среда**

Накратко, финализацията позволява да се почистват ресурси, свързани с даден обект, преди обектът да бъде унищожен от garbage collector. Обяснено най-просто, това е начин да се каже на CLR "преди този обект да бъде унищожен, трябва да се изпълни ето този код".

За да е възможно това, класът трябва да имплементира специален метод, наречен Finalize(). Когато garbage collector установи, че даден обект вече не се използва от приложението, той проверява дали обектът дефинира Finalize() метод. Ако това е така, Finalize() се изпълнява и на по-късен етап (най-рано при следващото преминаване на garbage collector), обектът се унищожава. Този процес ще бъде разгледан детайлно след малко. Засега просто трябва да запомните две неща:

* Finalize() **не може да се извиква явно**. Този метод се извиква само от системата за почистване на паметта, когато тя прецени, че даденият обект е отпадък.
* Най-малко **две** преминавания на garbage collector са необходими за да се унищожи обект, дефиниращ Finalize() метод. При първото се установява че обектът подлежи на унищожение и се изпълнява финализаторът, а при второто се освобождава и заетата от обекта памет. Всъщност в реалния живот почти винаги са необходими повече от две събирания на garbage collector поради преминаването на обекта в по-горно поколение.

### Деструкторите в C#

В .NET, класът System.Object дефинира Finalize() метод. Ако искаме да осигурим финализатор за нашия клас, бихме използвали следния код:

|  |
| --- |
| protected override void Finalize()  {  try  {  // Cleanup code goes here  }  finally  {  base.Finalize();  }  } |

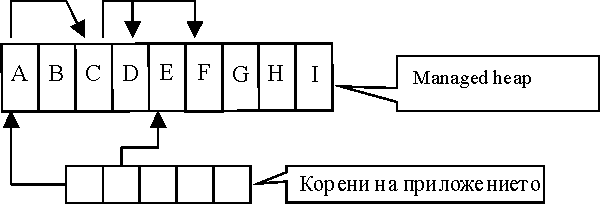
Както виждате, това, което правим, е да предефинираме Finalize() метода на класа System.Object (спомнете си, че всички типове в .NET наследяват System.Object). Използваме конструкцията try … finally за да се подсигурим, че независимо какъв е резултатът от изпълнението на почистващия код, ще бъде извикан Finalize() методът на родителския обект.

**35.Стратегии на управление на памет и събиране на боклук.Алгоритъм за събиране на боклук**

#### Алгоритъмът за почистване на паметта

Когато garbage collector започва своята работа, той предполага че всички обекти в managed heap са отпадъци, т.е. че никой от корените не сочи към обект от паметта. След това, системата за почистване на паметта започва да обхожда корените на приложението и да строи граф на обектите, достъпни от тях.

Нека разгледаме примера, показан на следващата фигура. Ако глобална променлива сочи към обект A от managed heap, то A ще се добави към графа. Ако A съдържа указател към C, а той от своя страна към обектите D и F, всички те също стават част от графа. Така garbage collector обхожда рекурсивно в дълбочина всички обекти, достъпни от глобалната променлива A:



Когато приключи с построяването на този клон от графа, garbage collector преминава към следващия корен и обхожда всички достъпни от него обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време на работата garbage collector се опита да добави към графа обект, който вече е бил добавен, той спира обхождането на тази част от клона. Това се прави с две цели:

* значително се увеличава производителността, тъй като не се преминава през даден набор от обекти повече от веднъж;
* предотвратява се попадането в безкраен цикъл, ако съществуват циклично свързани обекти (например A сочи към B, B към C, C към D и D обратно към A).

След обхождането на всички корени на приложението, Графът съдържа всички обекти, които по някакъв начин са достъпни от приложението. В посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следователно се считат за отпадъци. В нашия пример това са обектите B, G, H и I.

След идентифицирането на достъпните от приложението обекти, garbage collector преминава през хийпа, търсейки последователни блокове от отпадъци, които вече се смятат за свободно пространство. Когато такава област се намери, всички обекти, намиращи се над нея се придвижват надолу в паметта, като се използва стандартната функция memcpy(…). Крайният резултат е, че всички обекти, оцелели при преминаването на garbage collector, се разполагат в долната част на хийпа, а NextObjPtr се установява непосредствено след последния обект. Фигурата показва състоянието на динамичната памет след приключване на работата на garbage collector:



**38.Поколения в .NET среда.Управление на поколенията**

**Поколенията (generations)** са механизъм в garbage collector, чиято единствена цел е подобряването на производителността. Основната идея е, че почистването на част от динамичната памет винаги е по-бързо от почистването на цялата памет. Вместо да обхожда всички обекти от хийпа, garbage collector обхожда само част от тях, класифицирайки ги по определен признак. В основата на механизма на поколенията стоят следните предположения:

* колкото по-нов е един обект, толкова по-вероятно е животът му да е кратък. Типичен пример за такъв случай са локалните променливи, които се създават в тялото на даден метод и излизат от обхват при неговото напускане.
* колкото по-стар е обектът, толкова по-големи са очакванията той да живее дълго. Пример за такива обекти са глобалните променливи.
* обектите, създадени по едно и също време обикновено имат връзка помежду си и имат приблизително еднаква продължителност на живота.

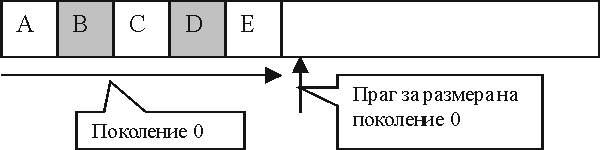
Много изследвания потвърждават валидността на изброените твърдения за голям брой съществуващи приложения. Нека разгледаме по-подробно поколенията памет и това как те се използват за оптимизация на производителността на .NET garbage collector.

#### Поколение 0

Когато приложението се стартира, първоначално динамичната памет не съдържа никакви обекти. Всички обекти, които се създават, стават част от Поколение 0. Казано накратко Поколение 0 съдържа новосъздадените обекти – тези, които никога не са били проверявани от garbage collector.

При инициализацията на CLR се определя праг за размера на Поколение 0. Точният размер на този праг не е от особено значение, тъй като може да се променя от garbage collector по време на работа с цел подобряване на производителността. Да предположим, че първоначално стойността на този праг е 256KB.

Следващата фигура показва състоянието на динамичната памет след като приложението е работило известно време. Виждаме, че са създадени известен брой обекти (всички част от Поколение 0), а обекти B и D вече са станали недостъпни (т.е. подлежат на почистване).



Да предположим, че приложението иска да създаде нов обект, F. Добавянето на този обект би предизвикало препълване на Поколение 0. В този момент трябва да започне събиране на отпадъци и се стартира garbage collector.

#### Почистване на Поколение 0

Garbage collector процедира по описания по-горе алгоритъм и установява че обекти B и D са отпадъци. Тези обекти се унищожават и оцелелите обекти A, C и E се пренареждат в долната (или лява) част на managed heap. Динамичната памет непосредствено след приключването на събирането на отпадъци изглежда по следния начин:



Сега оцелелите при преминаването на garbage collector обекти стават част от Поколение 1 (защото са оцелели при едно преминаване на garbage collector). Новият обект F, както и всички други новосъздадени обекти ще бъдат част от Поколение 0.

Нека сега предположим, че е минало още известно време, през което приложението е създавало обекти в динамичната памет. Managed heap сега изглежда по следния начин:

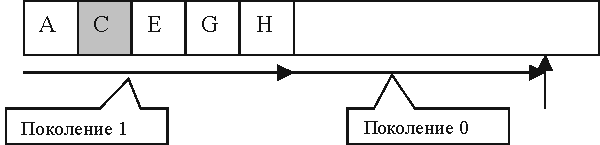


Добавянето на нов обект J, би предизвикало препълване на Поколение 0, така че отново трябва да се стартира събирането на отпадъци. Когато garbage collector се стартира, той трябва да реши кои обекти от паметта да прегледа. Както Поколение 0, така и Поколение 1 има праг за своя размер, който се определя от CLR при инициализацията. Този праг е по-голям от този на Поколение 0. Да предположим че той е 2MB.

В случая Поколение 1 не е достигнало прага си, така че garbage collector ще прегледа отново само обектите от Поколение 0. Това се диктува от правилото, че по-старите обекти обикновено имат по-дълъг живот и следователно почистването на Поколение 1 не е вероятно да освободи много памет, докато в Поколение 0 е твърде възможно много от обектите да са отпадъци. И така, garbage collector почиства отново Поколение 0, оцелелите обекти преминават в Поколение 1, а тези, които преди това са били в Поколение 1, просто си остават там.

Забележете, че обект C, който междувременно е станал недостъпен и следователно подлежи на унищожение, в този случай остава в динамичната памет, тъй като е част от Поколение 1 и не е проверен при това преминаване на garbage collector.

Следващата фигура показва състоянието на динамичната памет след това почистване на Поколение 0.



Както вероятно се досещате, с течение на времето Поколение 1 бавно ще расте. Идва момент, когато след поредното почистване на Поколение 0, Поколение 1 достига своя праг от 2 MB. В този случай приложението просто ще продължи да работи, тъй като Поколение 0 току-що е било почистено и е празно. Новите обекти, както винаги, ще се добавят в Поколение 0.

**№ 6 Обща класификация на класовете в MFC среда. Структура на Windows приложвние при използване на подкрека от MFC среда. Основни класове на приловение. Последователност при конструиране на класовете.**

###### Основни View класове към MFC

1. **CView - стандартен дъщерен прозорец за вашето изображение.**
2. **CScrollView - добавя scroll възможности към CView .**
3. CFormView- предостявя диалог за вграждане на контроли в изображението
4. **CRecordView- Производен на горния, използва се при работа с ODBC БД – изобразява запис от ODBC база данни.**
5. **CDaoRecordView- подобен на горния, но за работа с DAO .**
6. **CEditView- съдържа edit control, който попълва клиентската област .**
7. **CTreeView- съдържа tree view control, койо попълва клиентската област на изображението.**
8. **CRichEditView- съдържа rich edit control, който попълва клиентската област на изображението.**
9. **CListView- съдържа list control.**
10. **CHtmlView създава изгледи от html формати и други формати, поддържани от контрола WebBrouser на Microsoft Internet Explorer.**

**Ето наследствените връзки м/ду тези класове:**

**CView**

**CScrollView CCtrlView**

**CEditView CListView**

**CFormView**

**CRichEditView**

**CTreeView**

**CRecordView CDaoRecordView**

**CHtmlView**

1. Общи класове - това са групите класове за управление на файлове, изключителни ситуации и често използвани класове.
2. Класове, които управляват възможности (меню, графични обекти, контекстни устройства, прозорци, диалози).
3. Класове, които структурират приложение (инструменти свързани с условията, изображения, нишки и т.н.).
4. Класове колекции – служат като контейнери за съхраняване на други обекти.
5. OLE класове.
6. Database класове – 7-8 класа свързани с обработката на структурирани данни.
7. Контролите.
8. Window Socket класове – тук се включват класовете за работа с клиент-сървър приложения, работа в мрейа и т.н..

**№ 8 Методика на изграждане на работоспособно приложение във визуална среда. Пример с VISUAL C++. Основни типове класове в приложението.**

**Cdocument CView display**

**printer**

**archive** **data**

Едно изображение може да работи с множество документи, както и един документ – в много изображения.

Едно приложение във визуална среда има следните класове:

1. Документен клас (Cdocument) – в него систематизирана цялата обработка на данните.
2. Клас инображение (CView) – неговите методи са отговорни за изобразяване на данните, входа и изхода.
3. Клас Frame – това е най-голямото изображение. Това е мястото където се съдържа клиентската област на изображението
4. Клас на приложението – този клас обединява останалите и осъществява взаимодействието със средата (Windows).

Hello World във визуална среда. Използваме метода OnDrow:

*void CHelloView::OnDraw(CDC\* pDC){*

*CHelloDoc\* pDoc = GetDocument(); //указател към документа*

*ASSERT\_VALID(pDoc);*

*// добавен код*

*CRect rcClient;*

*GetClientRect( rcClient );*

*pDC->DrawText( "Hello World", -1, rcClient, DT\_SINGLELINE |DT\_CENTER | DT\_VCENTER );*

*}*

**№21 Езикови средства за създаване на усотйчив код. Предоставени от 32 битова среда възможности за целта. Въведение в SEH механизма. Терминираща обработка (\_\_finally). Възможни двусмислия при наличие на терминиращи блокове.**

**Въведение в SEH.**

SEH(Structure Exception Handling) структурирана обр. на изключителни ситуации.

Това е вграден механизъм за обработка на софтуерни и хардуерни изкл. ситуации.

Не добре структуриран код:

*int ConcStr( TCHAR\* pszDest, TCHAR\* pszSrc, int cDest )*

*TCHAR\* pResult = NULL;*

*if( pszDest && pszSrc ){*

*int nDest = lstrlen( pszDest);*

*int nSrc = lstrlen( pszSrc);*

*if(( nDest + nSrc) < cDest ){*

*pResult = lstrcat( pszDest, pszSrc );*

*}*

*}*

*if ( pResult ){*

*return lstrlen (pResult);*

*}*

*else{*

*return 0;*

*}*

*}*

Добре структуриран код:

*int ConcStr( TCHAR\* pszDest, TCHAR\* pszSrc, int cDest )*

*{*

*\_\_try*

*{*

*TCHAR\* pResult = lstrcat ( pszDest, pszSrc );*

*return lstrlen( pResult );*

*}*

*\_\_except( EXCEPTION\_EXECUTE\_HANDLER )*

*{*

*return 0;*

*}*

*}*

Exception ще се генерира във вторият случай.

1. При липса на …………

Блокът след exception handler ще се изпълни при наличие на exception в try. Блокът exception се изпълнява в обсега на try.

В 32 битова среда има следните възможности за прихващане и обработка на exception:

1. Може да се ползва SEH механизма на ОС.
2. Може да се ползва собствения за езика механизъм за обработка на exceptions.
3. Възможна е комбинация то 1 и 2.

При откриване на exception става:

1. Изпълняваната нижка се прекратява.
2. Управлението се предава от *usermode* в *kernelmode* (ядрото на ОС поема управлението).

Търси се начин за реакция чрез exception блок. Ако има то тоий се изпълнява, а ако няма диспечера генерира служебен exception handler.

Ако има блок exception handler то:

1. Да се изчисти паметта от излишни данни.
2. Да се възстановят операциите в БД (ако има).
3. Да се освободят заетите ресурси.
4. В LOG файл може да се запише информация.
5. Диалогов прозорец с полезна информация.
6. **Терминираща обработка (\_\_finally).**

*\_\_finally* представлява блок за който може да се гарантира че винаги ще се изпълни.

*BOOL Myfunc()*

*{*

*int\* p = 0;*

*\_\_try{ // използва се p }*

*\_\_finally{ delete p; } //освобождава паметта винаги*

*return fReturn;*

*}*

Ако има exception блок, то finally се изпълнява преди него.

finally се изпълнява във следните случаи:

1. След изход форсиран от try секция (чрез exit, return).
2. След прекратяване на try блока чрез exception.
3. При нормален изход.

Добрият стил изисква return да бъде изведен извън блоковете try и finally, но return може да е в try и finally. Ако return е в try блока връщаната стойност не се връща в този момент. Компилаторът отлага връщането като пази резолтата в междинна памет. Ако след finally блока има още един return, то неговата стойност няма да се върне, а ще се върне междинно съхраняваната.

**№ 22 Филтър за обработка на изключителни състояния (\_\_except). Събиране информация за състоянието при събитие.**

Exception filter е израз, който може да ……….. ……….. . Той се изчислява и стойността определя начина на реакция при случване на exception.

……………………………………………………………………….

EXCEPTION\_EXECUTE\_HANDLER – term.handler (1)

EXCEPTION\_CONTINUE\_SEARCH (0) – dispatcher ще търси друг.

EXCEPTION\_CONTINUE\_EXECUTION – пренебрегва (-1)

Първият има стойност 1:

Ще се изпълни блокът след except и управлението ще се подаде след блока except.

Вторият има стойност 0:

Той показва на ОС да пренебрагне нашият exception блок и да търси обхващащ exception блок.

Третият има стойност –1:

Той пренебрагва exception блока т.е. той не се изпълнява.

ОС изработва множество идентификатори, които указват причината за exception-а.

*\_\_except ( GetExceptionCode() == EXCEPTION\_ACCESS\_VIOLATION )*

*{*

*cout << “прихваната изкл. ситуация” <<endl;*

*} // опит за четене/запис без инициал. на указател или без съответни права.*

По-общ анализ:

*\_\_ except( ExceptionFilter( GetExceptionCode()))*

*където:*

*int ExceptionFilter( int nException )*

*{*

*int nReturn;*

*switch( nException ){*

*case EXCEPTION\_INT\_DIVIDE\_BY\_ZERO: nReturn = EXCEPTION\_EXECUTE\_HANDLER;*

*break;*

*………………….*

*default: nReturn = EXCEPTION\_CONTINUE\_SEARCH;*

*}*

*return nReturn;*

*}*

**№ 23 Вграден в С++ механизъм за реакция на изключителни събития (С++ Exception Handling). Стандартни обекти на exception библиотеката в ANSII С++ стандарт. Прихващане на изключителни състояния в етапа конструиране на обект – особености.**

*int Divide( int n1, int n2 )*

*{*

*int nReturn = 0;*

*try{*

*if( !n2 ) throw range\_error();*

*nReturn = n1 / n2;*

*}*

*catch ( range\_error& e ){*

*cout << “опит за / на 0” << endl;*

*}*

*return nReturn;*

*}*

В try блока се генерират изключителни ситуации. Може да се генерира дъщерна изключителна ситуация, а да се прихване родителска. Трябва try и catch да са в една функция. Ако ниама catch, то се търси в обхващащата функция дали има catch, ако никъде няма то се извиква terminate. При генериране на изключителни ситуации с throw автоматично се изпълняват деструкторите на всички обекти създадени от началото на try блока.

# Standart Exception Library

**exception**

**logic\_error runtime\_error**

**domain\_error (вътр. грешка) range\_error**

**invalid\_argument overflow\_error**

**length\_error**

**out\_of\_range**

*SomeFunc()*

*{*

*try{…………..*

*throw range\_error( “ невъзможност за заделяне достатъчно памет”);*

*……*

*}*

*catch( runtime\_error e){*

*cout << e.what() << endl ; // e.what() – стандартна (извиква съобщение).*

*}*

*}*

Интересен синтаксис е следният:

catch (…) *// прихваща за обработка всички exceptions.*

Важно е да се обхване в секцията кода на …………. Конструктор, тъй като не връща стойност т.е. не може да информира за успех или неуспех относно работата му.

**{ cout << “ …………sssdcg “ << endl;**

**throw; }}**

**№25 Съвместяване на двата механизма (Win32 SEH и C++ Exception Handling).**

Някои проблеми в SEH механизма:

1. Възможно е ако работим само със SEH механизма да пропуснем деструкцията на някои проблеми.
2. SEH механизма дава възможност за много точно детайлиране на причината на ниско ниво, но при положение че сме прихванали съответната причина при SEH.
3. Повечето стандартни С++ функции са написани така, че хвърлят С++ exception, а не SEH exception.

Възможно е рпеобразуването на обекти на SEH към С++ exception обекти. За целта в 32 битовите SEH механизми е преджидена \_set*\_*se\_translator позволява дефинирането на потребителска функция и извършва транслация на SEH механизма към С++ exception механизма. В My\_func се изпълнява транслацията и се хвърля (throw) С++ exception обект, който съдържа информация с exception.

*class CWin32Except{*

*unsigned int m\_nCode;*

*public: CWin32Except( unsigned int nCode ) : m\_nCode(nCode){};*

*unsigned int Code() const {return m\_nCode;};*

*}*

използване:

*void SEH\_MyFunc( unsigned int nCode, EXCEPTION\_POINTERS\* pExp ){*

*throw CWin32Except( nCode );*

*}*

*void DoAccessViolation(){*

*int\* p =0;*

*\*p = 32;*

*}*

*void main(void){*

***\_se\_translator\_function fn old;***

*fnold = \_set\_se\_translator( SEH\_MyFunc );*

*try{*

*DoAccessViolation();*

*}*

*catch( CWin32Except& e ){*

*cout << “ изкл. с CWin32Except ” << endl; cout << “Exception is “ << e.Code() << endl;*

*}*

*\_set\_se\_translator( fnold);*

*}*

**№ 27 Вход/изход и сериализация. Работа с файлови обекти без сериализация. MFC класове за целта. Цел и предимства за въвеждане на архивни обекти. Предефиниране на операциите за сериализация.**

## Работа без сериализация – базов клас CFile

*CFile myfile;*

*CFileException e;*

*if ( file.Open( \_T(“My.txt”), CFile::modeReadWrite, &e)){*

*// работим с файла*

*}*

*else{*

*………e.ReportError();*

*}*

същото може и така:

*try{*

*CFile file ( \_T(“MyFile.txt”), CFile::modeReadWrite);…..*

*}*

*catch( CFileException& e){*

*eReportError(); eDelete();*

*}*

* **затваряне:**

1. file.Close();
2. CFile обект се затваря авт. при излизане извън обсег.

* **четене/ запис**

*BYTE buff[0x4000]; CFile file(………………);*

*DWORD length = file.GetLenght();*

*while(length) {*

*UINT nByteRead = file.Read(buff, sizeof(buff));*

*lenght -= nByteRead;*

*}*

- file.Write(buff, nByteRead); // записва определен брой байтове от буфера

- file.Seek( относит. отместване в байтове, спрямо какво)

- препоръчително е четенето да се обхване oт try catch

* изтриване (Remove())
* преименоване на файлове (Rename())

# Производни на CFile класове

**CMemFile, CSharedFile**

COleDataObject::GetFileData

CSocketFile, стои между архивния обект и CSocket обекта.

CStdioFile , наследник на CFile. Пример:

пример:

*try {*

*CStdioFile file( \_T(“My.txt”), CFile::modeRead); …..*

*}*

*catch( CfFileException\* e) { }*

CInternetFile, CGopherFile и CHttpFile .

# Изброяване на файлове и директории

::FindFirstFile

:: FindNextFile()

:: FindClose()

**Универсален подход на I/O – използване на архиви (базов клас CArchive)**

Ето пример: искате в отоворен файл да запишете 2 променливи a,b:

*file.Write( &a, sizeof(a));*

*file.Write( &b, sizeof (&b));*

Ето другият подход:

*CArchive ar( &file, CArchive::store);*

*ar << a << b;*

Архива е един междинен обектмежду нас и запомнящата среда, скриващ спецификацията на запомнянето, а представящ на нас само стандартният диск. Едни и същи операции за вход/изход да бъдат използвани за данни с различен тип. Архива е винаги между нашето прилижение и запомнящата среда. Функцията на междинният слой е като един транслатор. Най-често архива се асоцира с диск. Първо създаваме архива – обект, асоцираме го със запомнящата среда. Командите за вход/изход трябва да се предефинират за нашите данни и обекти. Всички примитивни типове имат предефинирани в MFC операции за << >> (BYTE, WORD, LONG, DWORD, float, double, intchar, char, unsigned int). Оперторите са предефинирани и за непримитивни типове, за които има стандартни MFC класове:

пример: *Cstring string; ar<<string;*

също и за: Ctime, Crect, Csize, ColeDateTime, cole variant …, както и за структури SIZE, POINT, RECT.

Ако се създават собствени класове и ако искаме те да поддържат всички обекти от тези класове и ако направим класа наследник на Cobject тогава не е нужно да предефинираме командите за вход/изход.

**№ 28 Създаване на собствени класове, поддържащи сериализация. Кога се налага пряко използване на Serialize().**

**Създаване на клас, поддържащ сериализация.**

* Новосъздаденият клас наследява CObject;
* В декларацията на класа DECLARE\_SERIAL( име на класа)
* Предефинирвате Serialize() на базовия клас и сериализирате данновите членове на производния клас (ще поясним по-късно).
* Подразбиращ се конструктор на класа.
* В частта имплементация на класа IMPLEMENT\_SERIAL

Конструктор по подразбиране – това е конструктор без параметри. Когато класът поддържа сериализация този конструктор задължително трябва да го има. Донамичното създаване на обектиизползва винаги подразбиращият конструктор.

>> оператор за десериализиране, <<оператор за сериализиране.

**Кога се използва Serialize().**

* При десериализиране на конструиран обект, не знаем името и сме заделили памет.
* Ако обектът е сериализиран със Serialize() трябва да се десериализира със Serialize().
* Ако създаваният клас има даннов елемент проявление или наследник на класа CОbject също се използва Serialize(), защото операциите << >> са дефинирани в CОbject да работят с указатели на обекти, а не с обекти.
* Ако сериализираният клас има даннов елемент указател към Cоbject или негов наследник, тогава имаме 2 възможности 1. Първо трябва да се конструира данновият обект във функцията за сериализация и след това да се използва указателя. Използването става по 2 начина:

PObject // указател

* PОbject ->Serialize();
* ar>>m.mydata>>p.Object

IMPLEMENT\_SERIAL – този макрос иска три параметъра.

пример:

class CLine : public CObject

{ DECLARE\_SERIAL (CLine)

protected: CPoint m\_ptFrom;

CPoint m\_ptTo;

public: CLine() {}

CLine( CPoint from, CPoint to) {m\_ptFrom = from; m\_ptTo = to;}

void Serialize (Carchive& ar ); };

като дефинирате ф-ията Serialize така:

void CLine::Serialize( CArchive& ar)

{ CObject::Serialize( ar);

if ( ar.IsStoring()) ar << m\_ptFrom << m\_ptTo;

else ar >> m\_ptFrom >> m\_ptTo;

}

добавяте и някъде в имплементацията:

IMPLEMENT\_SERIAL( CLine, CObject, 1)

**IMPLEMENT\_SERIAL( CLine, CObject, 1 | VERSIONABLE\_SCHEMA)**

**В примера са спазени всички правила за сериализация описани по-горе**

**№ 29 Версии при сериализация. Работа с указатели към обекти и проблеми при използване на псевдоними на обекти при сериализация.**

Когато работим с различни версии ни трябва да предвидим код в Serialize().Интересна е и още една особеност: Предефинирането работи за указатели към такива обекти-наследници, а не за самите обекти. Така че:

*CLine\* pLine = new CLine( CPoint( 0,0), CPoint( 40,40));*

*ar << pLine; //работи добре*

*CLine line( CPoint( 0,0), CPoint( 40,40));*

*ar << line; // няма да работи*

**Aко искаме да сериализираме по стойност, а не по указател, е добре така:**

CLine line( CPoint( 0,0), CPoint( 40,40));

ar << &line; // ще работи за сериализация

при десериализация:

CLine\* pLine;

ar >> pLine; //всичко е добре

CLine line = \*pLine; // операция “=” е предеф. в CLine като копиране

delete pLine; // вече ненужен

**№ 34. Memory-mapped файлове като средство за пестене на ОП и създаване на общи блокове. Избягване на фрагментацията при често алокиране на памет, пестене на пространство от swap файла при работа с инициализирани данни.**

Memory mapped file – средство за оптимално управление на паметта. Състой се от с асоцииране на файла или част от файл с ОП. Файла или част от файл се асоциира с виртуалното адресно пространство и тогава над него могат да се правят определени операции без да се качва.

*HANDLE hFile=::CreateFile(...)//създаваме Handle*

*HANDLE hMap=::CreateFileMapping(hFile,...);*

*LPVOID IpvFile=::MapViewoffile(hMap,...);//мапва целия файл*

*DWORD dwFilesize=::GetFileSize(hFile,...)//използваме файл*

*::UnmapViewofFile(IpvFile);*

*::CloseHandle(hMap);*

*::CloseHandle(hFile);*

Два процеса могат да ползват общ *hMap* т.е. те имат обща памет:

*GlobalAlloc(...,GMEM\_SHARED...);*

В Win32 не прави shared блок, както в Win16. Обща памет, но не общ файл както по – горе без *CreateFile(...)* и с подаване на параметри 0хFFFFFFFF вместо hFile.

*GlobalAlloc* е наследена в Win16. Тук в 32 битовата среда всъщност не действа , но може да се запише.

Съвет при работа с динамична памет:

- хипът се фрагментира при продължителна работа. Решението е по – често да се унищожава и да се създава нов или да се прекомпонира.

- викайте heapmin в NT за освобождаване на големи блокове

- не викайте HeapFree за малки блокове заделени с new.

- стекът вече не е ограничен до 64К и става толкова голям, колкото е необходимо.

Как да пестим пространство от swap файла?

- EXE и DLL файлове не са в swap file – те се включват във виртуалното адресно пространство на всеки процес. Добре е и константните данни по някакъв начин да се прикачат към тях, за да не попадат в swap file.

Низове

Ако те са непроменяеми за цялото изпълнение , декларирайте const *charmystr[]=”my string”;*

Низът се съхранява заедно с кода – в секцията за инициализирани константни данни . тя се съхранява заедно с EXE файла.

В тази секция не можем да слагаме само С++ обекти, създадени чрез конструктор. Например:

*const Crect my\_rect(0,0,100,100);* се поставя в .bss, който се вкарва в swap file и всеки процес има копие на този обект. По – лошо е:

*const CString my\_str(“new instance”);*  което води до :

1. поставяне на *CString* обект в .bss секцията
2. масивът от символите в .data (иниц. неконст. данни )
3. заемане памет за копие на символите за всеки стартиран процес.

Нищо не попада в EXE и всичко харчи памет.

**20.Таймер. Работа с таймери.Пропъртита, събития.Пример**

**4. Таймер**

Ето един фрагмент от програма, дефиниращ 2 таймера:

#define TIMER\_SEC 1

#define TIMER\_MIN 2

.....

SetTimer(hwnd, TIMER\_SEC, 1000, NULL);

SetTimer(hwnd, TIMER\_MIN, 6000,NULL);

....

case WM\_TIMER :

switch(wParam) / /съдържа идентификатор на таймер

{case TIMER\_SEC : // обработка веднаж в секунда;

break;

case TIMER\_MIN : // обработка веднаж в минута;

break;

#### Таймери в 32 битови среди

**CWnd:: SetTimer() CWnd::KillTimer();**

**MFC макросът ON\_WM\_TIMER насочва WM\_TIMER към CMainWindows::OnTimer(UINT nTimerID);**

**1.Приложението с таймер би следвало да изглежда така:**

***BEGIN\_MESSAGE\_MAP ( CMainwindow, CFrameWindow)***

***ON\_WM\_CREATE()***

***ON\_WM\_TIMER()***

***END\_MESSAGE\_MAP***

***int CMainWindow::OnCreate(…………)***

***{…………if(!SetTimer(ID\_TIMER\_XX,100,NULL)) {..}***

***return 0;}***

***void CMainWindow::PnTimer(UINT nTimerID)***

***{ // например изрисува нещо през 100 мс }***

**14.SDI и MDI приложения. Структура и пример**

Разлики между SDI и MDI:

1. В MDI има повече от един отворен документ, докато в SDI , за да отвори втори трябва да затворим първия.
2. В MDI могат да се поддържат различни типове документи.
3. При MDI в менюто има опция Windows за превключване на прозорците.
4. MDI има поне 2 менюта , а SDI има едно .Първото при отворен документ, а второто при затворен.
5. В SDI има една рамка, а в MDI има главна и дъщерна рамка.

Множество изгледи под един документ в MDI приложения

Използва се многодокументния шаблон-CMultiDocTemplate.Тъй като документа е същия не се създава нов документ, а само нов изглед.Т.е. документа и рамката си остават същите.MFC осигурява списъчна структура за обхождане на всички изгледи(като например при UpdateAllView).Всеки изглед вика Get Document, за да получи данни.Ако в някое View променим данните и искаме промените да се отразят в други View, налага се да се използва UpdateAllView, който пък вика OnUpdate на всеки View.Този метод не се вика автоматично-ние трябва да го осигурим. UpdateAllView обхожда всеки View и ги обновява, като инвалидизира (Invalidate) целия прозорец.Ако искаме оптимизация ние трябва да променим реализацията на UpdateAllView и по-точно да променим параметрите, с които се вика OnUpdate.

# Main Frame Windows

**-- File ….. X**

**в SDI**

**CView**

**CMainFrame**

**-- File ….. X**

#### File X

**в MDI**

### CChildFrame

**CMainFrame**

# Време на живот на документа и изображението

**SDI MDI**

**Construction Creates Additional documents**

**OnNewDocument Construction**

**Cteate addit.**

**Documents OnNewDocument**

**OnCloseDocument**

**OnCloseDocument**

**Destruction**

###### Destruction

## **7.Съобщения и карти на съобщения. Макроси свързани с картите на съобщения.Последователност при търсене на съответствие**

## Съобщения и карта на съобщения

**On\_MESSAGE(WM\_MYMESSAGE, OnMyMessage)**

**afx\_msg LPRESULT OnMyMessage(WPARAM wParam,LPARAM lParam);**

**message map структури \_messageEntries масиви member ф-ии**

**CWnd**

**&CcmdTarget::messageMap**

**&CWnd::\_messageEntries[0] WM\_DRAWITEM CWnd::OnDrawItem**

**….**

**CFrameWnd**

**&CWnd::MessageMap**

**&CFrameWnd::\_messageEntries[0]**

**WM\_MENUSELECT**

**CFrameWnd::OnMenuSelect**

### WM\_CREATE CFrameWnd::OnCreate

**WM\_DESTROY CFrameWnd::OnDestroy**

### ChelloWnd ….

**&CFrameWnd::messageMap**

**&CHelloWnd::\_messageEntries[0] WM\_PAINT CHelloWnd::OnPaint**

### Кодови таблици и макрос \_Т

***“Hello” -ANSI***

#### L”Hello” -Unicode

***\_T(“Hello”) - според директива #define \_UNICODE***

**забележка: 1. декларации TCHAR вместо char; 2. TCHAR\*или LPTSTR или LPCTSTR вместо char\* ;3. дължината на знак е sizeof(TCHAR); 4. вместо стандартни run-time ф-ии: strcopy() tcscpy() или tcscat().**

**CDocument**  **CView** **display**

**printer**

**archive**  **data**

# Същото приложение във визуална среда

**void CHelloView::OnDraw(CDC\* pDC)**

**{ CHelloDoc\* pDoc = GetDocument(); //указател към документа**

**ASSERT\_VALID(pDoc);**

**// добавен код**

**CRect rcClient; GetClientRect( rcClient );**

**pDC->DrawText( "Hello World", -1, rcClient, DT\_SINGLELINE |**

**DT\_CENTER | DT\_VCENTER ); }**

**18.Windows Presentation Foundation (WPF).XAML.Контроли и логическо дърво.Примери**

XAML - Extensible Application Markup Language

* SXAML code is short and clear to read
* eparation of designer code and logic
* Graphical design tools like Expression Blend require XAML as source.

XAML example:

**19.LINQ-Language Integrated Query.Query Expressions.Ламда изрази**

Основни фундаменти на LINQ:

* LINQ - използвани езици

C# 3.0

VB 9

* Особености
  + Lambda Expressions
  + Query Expressions
    - Delegate functions
    - Type inference
    - Anonymous types
    - Extension methods
    - Expression trees

Query Expressions:

* Introduce SQL-Like Syntax to Language
* Compiled to Traditional C# (via Extension Methods)

**from** *itemName* **in** *srcExpr*

**join** *itemName* **in** *srcExpr* **on** *keyExpr* **equals** *keyExpr*

(**into** *itemName*)?

**let** *itemName***=** *selExpr*

**where** *predExpr*

**orderby** (*keyExpr* (**ascending | descending**)?)\*

**select** *selExpr*

**group** *selExpr* **by** *keyExpr*

**into** *itemName query-body*

Lambda Expressions Predicates:

* + Predicate
    - (p) => p.Gender == “F”
  + Projection
    - (p) => p.Gender ? “F” : “Female”
    - “Each person *p* becomes string “Female” if Gender is “F””

**13.Диалози-стандартни и потребителски. Видове и приложение.Пример за употреба**

Стандартни диалогови кутии:

* Класът MessageBox позволява извеждане на стандартни диалогови кутии:
  + съобщения към потребителя
  + въпросителни диалози

**16.GDI+.Методи за изчертаване на основните фигури.Запълване, контур и специални ефекти**

**GDI+: Graphics Objects**

Координати:

Pen:

Brush:

Drawing a rectangle:

You can override OnPaint event of your form to draw an rectangle. The LinearGradientBrush encapsulates a brush and linear gradient.

protected override void OnPaint(PaintEventArgs pe)

{

Graphics g = pe.Graphics ;

Rectangle rect = new Rectangle(50, 30, 100, 100);

LinearGradientBrush lBrush = new LinearGradientBrush(rect, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);

g.FillRectangle(lBrush, rect); }

**Drawing an Ellipse:**

An ellipse( or a circle)  can be drawn by using DrawEllipse method.

This method takes only two parameters, Pen and rectangle.

**protected override void OnPaint(PaintEventArgs pe)   
{   
Graphics g = pe.Graphics ;   
Pen pn = new Pen( Color.Blue, 100 );   
Rectangle rect = new Rectangle(50, 50, 200, 100);   
g.DrawEllipse( pn, rect );**

**17.Вход/Изход в .NET. Работа с файлове, директории,потоци, четци и писци**

Какво представляват потоците?

Потоци – пример:

Файлови потоци:

Четци и писачи:

Класовете File и FileInfo:

**51.NET Framework и система за управление на общи типове. Типовете в CLR**

NET Common Language Runtime (CLR) включва Общата система на типовете (Common Type System (CTS)), която дефинира типовете данни, поддържани от CLR.

### **CTS описва .NET типовете**

Типовете данни в CTS биват най-разнообразни:

* примитивни типове (primitive types – int, float, bool, char, …)
* изброени типове (enums)
* класове (classes)
* структури (structs)
* интерфейси (interfaces)
* делегати (delegates)
* масиви (arrays)
* указатели (pointers)

**32.Управление на паметта в Windows**

**Организация на адресното пространство:**

**съдържание**

**0 – 64KB служебни**

**Над 64KB За модули на изпълнимия файл**

**Над горния За heaps и threads stacks**

**Над тях За DLL**

**Над тях Системни DLL:Kernel32,User32**

**2GB – 4GB За нуждите на ОС**

Резервирането заделя блок с определена големина:

pMem = VirtalAlloc(<начален адрес на блока NULL>,<бр.страници на резервиране>,<права на достъп>):

За виртуалната памет се прави файл върху диска, който съдържа страници. Обмена м/у вирт. и ОП памет става по страници.Един процес може да има най много 4КВ обем страници. Една страница е валидна ако е в ОП.

Ако страницата я няма в ОП , тя се намира в swap файла и се качва в паметта. Ако не достига памет, то страницата се сваля от ОП, но всъщност тя се припокрива. Има доста механизми и особености при припокриване.

Заделянето на памет за нуждите на процес е двустъпков процес:

Резервиране Reserved

Ангажиране - Commit

Смисълът е да се отложи за колкото се може по късен момент заделянето на памет, но възможно непосредствено преди записване.

На етапа Commit паметта е заделена в Paging файла, а в паметта се заделя в ОП, едва когато се опитаме да запишем.

**31.Run-time сериализация**

**55.Проектиране на тип, предлагащ събитие.Проектиране на тип. използващ събитие.Същностт на нещата**

**62.CLR поддръжка за делегати**

**58.Интерфейсни типове.Използване на интерфеси със стойностни типове**

**53.Елементи на типа:методи,събития,полета,properties**

**10.Добавяне на манипулатор на събития.Манипулатори на събития или On … методи**

**1.Въведение в програмирането в среда Windows. Вход чрез опащки и съобщения**

**31.Run-time сериализация**

**55.Проектиране на тип, предлагащ събитие.Проектиране на тип. използващ събитие.Същностт на нещата**

**62.CLR поддръжка за делегати**

**58.Интерфейсни типове.Използване на интерфеси със стойностни типове**

**53.Елементи на типа:методи,събития,полета,properties**

**10.Добавяне на манипулатор на събития.Манипулатори на събития или On … методи**

**1.Въведение в програмирането в среда Windows. Вход чрез опащки и съобщения**

**53. Елементи на типа: методи, събития, полета, properties. Примери.**

**Методи** – функции, приемащи аргументи и връщащи стойности. Методите могат да бъдат статични и нестатични. Особеното при статичните методи е, че те принадлежат на самия клас, а не на обект от този клас. За да се извика метод дефиниран като static, трябва да бъде предшестван от името на класа. За да се извика нестатичен метод, трябва да бъде предшестван от името на обекта – инстанцията на класа, в който този метод е дефиниран.

Пример за извикване на статичен метод: Console.Writeline(…);

Пример за извикване на нестатичен метод: mydate.DayOfYear();

Пример за извикване на метод в .NET : MessageBox.Show(“Hello world!”);

MessageBox произлиза от Object и по този начин наследява няколко метода, имплементирани от Object. Единствения метод който MessageBox имплементира сам е метода Show(). Това е статичен метод и съществува в 12 различни версии.

Пример : DialogResult Show(string strText)

**Събитията** са членове на класовете заедно с конструкторите, полетата, методите и свойствата. Когато дадена програма дефинира метод за обработка на събитие, методът се нарича манипулатор на събитие (event handler). Аргументите на манипулатора съответстват на дефиниция на прототип на функцията, наречен делегат.

Пример за обработка на събитието Click на бутон

Инсталиране на манипулатор на събитието Click

btn.Click += new EvetHandler(MyButtonOnClick);

void MyButtonOnClick(object obj, EventArgs ea)

{

BackColor = Color.Blue;

}

**Свойствата** представляват блокове от код с get и set аксесоари.

Примери за своиства на контроли:

Свойства на контрола Button

btn.Text = “Click Me !”;

btn.Location = new Point(20, 20);

btn.Size = new Size(100, 20);

btn.BackColor = Color.Red;

btn.ForeColor = Color.White;

**58.Интерфейсни типове. Използване на интерфейси със стойностни типове.**

- Целта е не само функционална наследяемост (inheritance), но и възможност за поделяне на общ външен интерфейс между несвързани обекти.

- интерфейсния тип е частична спецификация на тип

- поддържа се наследяемост на интерфейсни типове

- интерфейсният тип съдържа

- методи(статични и на инстанция)

- полета( статични)

методите на интерфейса са public, abstract и virtual.

- Използване на интерфейси с Value типове

- можем да викаме инт. метод директно през value тип

- ако викаме value-type метод през интерф. указател(който е в хипа),

минаваме през пакетиране и прехвърляне на heap

**62. CLR поддръжка за делегатите.**

Платформата .NET Framework на Microsoft въвежда много нови концепции, технологии и термини. В сърцето на тази платформа е средата за изпълнение на общ език - Common Language Runtime(CLR).   
 CLR управлява изпълнението на .NET кода и осигурява услуги, улесняващи процеса на разработка.Компилаторите и инструментите правят достъпна функционалността на средата.

Възможностите на CLR са достъпни за всички езици, които компилират за нея.Ако CLR използва изключения, за да съобщава за грешки, тогава всички езици получават съобщенията за грешки посредством изключения. Ако CLR ви позволява да създадете нишка, тогава всички езици могат да създават нишки. Това означава, че би трябвало да изберете такъв език за програмиране, който ще ви позволи най-лесно да реализирате вашите намерения.Можете да разработвате вашия код на какъвто език искате, стига използвания от вас компилатор да компилира за CLR.

-CLR автоматично управлява паметта.

-Динамично-заделените обекти се разполагат в т. нар. Managed Heap

-Неизползваните обекти се почистват автоматично от т. нар. Garbage Collector

-Някои от най-неприятните проблеми в програмирането са почти невъзможни:

Загуба на памет (memory leaks)

-Достъп до освободена или неинициализирана памет

-В управлявания код няма указатели!

-Използват се референции към обекти

*Как CLR и компилаторът осигоряват поддръжката за делегати*

*Нека имаме в кода следния оператор:*

public delegate void Feedback (Object value, Int32 item, Int32 numItems);

*Компилаторът дефинира клас за делегата:*

public class Feedback:System.MulticastDelegate *//класът е public, защото делегатът е* *public*

{

public Feedback (Object target, Int32 methodPtr); *//конструктор*

public void virtual Invoke( Object value, Int32 item, Int32 numItems); *//методът има същия прототип като делегата*

*//следват методи осигоряващи асинхронното повикване на callback метода*

public virtual IAsyncResult BeginInvoke(Object value, Int32 item, Int32 numItems, AsyncCallback callback, Object object);

public virtual void EndInvoke(IAsyncResulet result);

}

-делегатът е клас, като всеки друг клас.

-делегатният обект е обвивка около метод и обект, с който методът работи.

-обръщението къмcallback метод е като с ф-я.Всъщност това е променлива, реферираща делегатен обект.

**11. Windows Forms в .NET. Контроли и йерархия на графичните конролите. Създаване на дъщерни форми и контроли. Пример.**

Windows Forms е стандартната библиотека на .NET Framework за изграждане на прозоречно-базиран графичен потребителски интерфейс (GUI) за настолни (desktop) приложения. Windows Forms дефинира набор от класове и типове, които позволяват изграждане на прозорци и диалози с графични контроли в тях, чрез които се извършва интерактивно взаимодействие с потребителя.

При настолните приложения, графичният потребителски интерфейс позволява потребителят директно да взаимодейства с програмата чрез мишката и клавиатурата, а програмата прихваща неговите действия и ги обработва по подходящ начин.

Windows Forms съдържа богат набор от стандартни контроли: форми, диалози, бутони, контроли за избор, текстови полета, менюта, ленти с инструменти, статус ленти и много други. В допълнение към стандартните контроли Windows Forms позволява разработчиците по лесен начин да създават допълнително свои собствени контроли, които да използват като части от своите приложения.

## **Йерархия на класовете**

На клас-диаграмата по-долу е показана част от класовата йерархия на библиотеката Windows Forms:

**61. Вътрешно представяне на делегат.**

Делегатите представляват .NET типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип.

Делегатите приличат на указателите към функции в C и C++ – съдържат силно-типизиран указател (референция) към метод. Те са структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура. Чрез тях се осъществяват "обратни извиквания" (callbacks). Могат да сочат както към статични методи, така и към методи на инстанция.

Пример:

public delegate void SimpleDelegate(string aParam);

class TestDelegate

{

public static void TestFunction(string aParam)

{

Console.WriteLine("I was called by a delegate.");

Console.WriteLine("I got parameter {0}.", aParam);

}

public static void Main()

{

*// Instantiation of а delegate*

SimpleDelegate simpleDelegate =

new SimpleDelegate(TestFunction);

*// Invocation of the method, pointed by a delegate*

simpleDelegate("test");

}

}

Делегатите в .NET Framework са специални класове и могат да бъдат два вида:

Единични (single-cast) делегати

Съдържат референция към един единствен метод;

Наследяват класа System.Delegate;

Множествени (multicast) делегати

Съдържат свързан списък от референции към методи;

Наследяват класа System.MulticastDelegate;

В C# могат да се декларират само Multicast делегати (чрез запазената дума delegate).

При извикване на multicast делегат, се изпълняват последователно един след друг всички методи от неговия списък. Ако multicast делегат връща стойност или променя ref или out параметър, резултатът е само от последния извикан метод от списъка с методи на делегата

Ако при извикване на multicast делегат някои от методите в неговия списък хвърли изключение, следващите методи от списъка не се извикват. На практика single-cast делегати почти не се използват и под делегат обикновено се има предвид multicast делегат.

Класът System.MulticastDelegate:

Е наследник на System.Delegate и е базов клас за всички делегати в C#. Съдържа метод Combine за сливане на списъците от методи на няколко делегата от еднакъв тип. Съдържа метод Remove за премахване на метод от списъка за извикване. Има метод GetInvocationList(), който връща масив от делегати – по един за всеки от методите в списъка за извикване на делегата. Има свойство Method, което описва сигнатурата на методите в делегата. Свойство **Target** което свързва обекта или обектите, чиито метод или методи ще се викат

**37 Модел на явна финализация в .NET среда.Интегриране на Finalize() и Dispose()**

Инстанциите на референтните типове в .NET се съхраняват в динамичната памет (в т. нар. managed heap).

При "препълване" на динамичната памет се включва системата за почистване на паметта (т. нар. garbage collector). Системата за почистване на паметта анализира динамичната памет и освобождава тези обекти от нея, които не се използват от програмата. При освобождаване на ненужните обекти им се извикват т. нар. finalizers, които освобождават използваните ресурси.

Предимства на автоматичното управление на паметта с помощта на garbage collector:

-не трябва ръчно да се освобождава паметта като в C++

-не се получава "изтичане на памет" (memory leaks) – много неприятен проблем

-не е възможно четене и писане по освободена памет или повторно освобождаване

-ресурсите винаги се освобождават

-паметта се заделя много бързо

Недостатъци:

-някои ресурси трябва да се управляват ръчно

-няма гаранция кога се изпълнява garbage collector-ът и колко време отнема;

Почистване на паметта (garbage collection) в .NET Framework:

* 1. Активира се при създаване на нов обект, когато има недостиг на памет
  2. Изчакват се всички нишки да достигнат безопасно състояние и се приспиват
  3. Намират се използваните от всички нишки обекти и всички обекти, достижими от тях
  4. Получава се граф на използваните обекти
  5. Идентифицират се ненужните обекти (тези, за които е установено, че не се използват)
  6. Ненужните обекти, които изискват финализация (т.е. имат да освобождават ресурси) се преместват специална опашка
  7. Всички останали ненужни обекти се освобождават
  8. Динамичната памет се пренарежда, така че лявата част да е заета памет, а дясната част – свободна (премахват се получените "дупки")
  9. Референциите към всички преместени обекти се пренасочват към новото им местоположение в динамичната памет
  10. Възобновява се работата на всички нишки

Някои обекти освен памет използват и други ресурси (например връзки към БД, файлови манипулатори, сокети и др.)

Финализацията осигурява правилното освобождаване на ресурсите, използвани от обектите

Всеки наследник на System.Object може да дефинира метод за финализация Finalize() (или деструктор в C#)

Методът Finalize() се извиква преди даден обект да бъде унищожен от системата за почистване на паметта

Освен динамичната памет CLR поддържа още две структури:

-Finalization Queue – опашка с обекти, които имат метод за финализация, но все още се използват от програмата

-Freachable Queue – опашка с обекти, които не се използват от програмата, но чакат да изпълнят метода си за финализация

При почистване на паметта обектите, които имат метод за финализация, не се освобождават веднага

Вместо това се прехвърлят от Finalization Queue във Freachable Queue

Финализицията се извършва от отделна нишка за всеки обект от Freachable Queue

В C# методът Finalize() от System.Object не може да се имплементира явно

Вместо това се използват деструктори

C# компилаторът преобразува деструкторите по следния начин:

~MyClass

{

*// Perform some*

*// cleanup here*

}

protected override void Finalize()

{

try

{

*// Perform some cleanup here*

}

finally

{

base.Finalize();

}

}

По време на финализация е възможно съживяване (resurrection) на обекти

Ако се добави референция от главната програма към обект, който се финализира, той се съживява

Garbage collector-ът не почиства съживени обекти, докато не останат неизползвани

Съживените обекти не се финализират повече, освен ако не се извика методът GC.ReRegisterForFinalize()

Използването на съживяване не е препоръчителна практика

Като правило не трябва да се разчита на финализацията защото:

- не е ефективна – обектите, които се финализират, се унищожават с най-малко 2 преминавания на garbage collector-а

- ресурсите се заемат ненужно дълго време

Ако все пак се използва финализация, трябва да се внимава:

* + по време на Finalize() да не се извикват други обекти, които изискват финализация
  + редът на изпълнение на отделните финализации е неопределен
  + възможни са синхронизационни проблеми – финализацията се изпълнява в отделна нишка
  + финализацията не трябва да е времеотнемаща

Използването на Finalize() не гарантира в кой момент ще се освободят ресурсите

Интерфейсът IDisposable позволява изрично (ръчно) освобождаване на ресурси

Ръчното управление на ресурсите е по-ефективно

Методът IDisposable.Dispose() трябва да освободи използваните ресурси и да извика GC.SuppressFinalize()

Препоръчва се винаги IDisposable и Finalize() да се имплементират заедно.

**32. Управление на паметта в Windows. Защита на пространството, виртуална памет, разширяване на оперативната памет към диск, валидизиране на страници, охраняеми страници, shared блокове, динамична резервация и ангажиране на блокове памет. Copy-on-write механизъм.**

Различните езици и платформи управляват паметта и ресурсите по различен начин:

-В езиците C и C++ паметта и ресурсите се заделят и освобождават ръчно от програмиста

-В езика C++ паметта и ресурсите на обектите се заделят при създаване на обект и се освобождават при унищожаване на обект (ръчно или при излизане на обекта от обхват)

-C++ използва деструктори за освобождение на ресурсите, използвани от обектите

В езици като Java, Perl, C# и VB.NET паметта се управлява напълно автоматизирано от т. нар. garbage collector

-Инстанциите на референтните типове в .NET се съхраняват в динамичната памет (в т. нар. managed heap)

-При "препълване" на динамичната памет се включва системата за почистване на паметта (т. нар. garbage collector)

-Системата за почистване на паметта анализира динамичната памет и освобождава тези обекти от нея, които не се използват от програмата

-При освобождаване на ненужните обекти им се извикват т. нар. finalizers, които освобождават използваните ресурси.

Размесване на сегментите в Windows програма:

-DOS програмите съдържат в началото си код от вида:

mov ax, DGROUP

mov ds, ax

Освен това компилаторът вмъква „пролог” и „епилог” части:

push bp

mov bp, sp

sub sp, x

Прологът и епилогът на всяка Windows far функция:

push ds mov bp,sp

pop ax push ds

nop mov ds,ax

inc bp sup sp,x

push bp

**12. Опционални и списъчни контроли. Основни пропъртита и събития. Приложение – пример.**

Контролите в Windows Forms са текстовите полета, етикетите, бутоните, списъците, дърветата, таблиците, менютата, лентите с инструменти, статус лентите и много други. Windows Forms дефинира базови класове за контролите и класове-наследници за всяка контрола. Базов клас за всички контроли е класът **System.Windows.Forms.Control**. Пример за контрола е

например бутонът (класът **System.Windows.Forms.Button**).Всяка контрола обработва собствените си **събития**Когато главната нишка на Windows Forms приложение получи съобщение,свързано с някоя от неговите форми, тя препраща съобщението до обработчика на съобщения на съответната форма. Този обработчик от своя страна проверява дали съобщението е за самата форма или за някоя нейна контрола. Ако съобщението е за формата, то се обработва директноот съответния обработчик на събития. Ако съобщението е за някоя от контролите във формата, то се предава на нея. Контролата, която получи съобщението, може да е обикновена контрола или контейнер-контрола.Когато обикновена контрола получи съобщение, тя го обработва директно. Когато контейнер-контрола получи съобщение, тя проверява дали то е за нея или е за някоя от вложените контроли. Процесът продължава,

докато съобщението достигне до контролата, за която е предназначено.Класът **System.Windows.Forms.Form** е базов клас за всички форми вWindows Forms GUI приложенията. Той представлява графична форма - прозорец или диалогова кутия, която съдържа в себе си контроли и управлява навигацията между тях. Повечето прозорци имат рамка и специални бутони за затваряне, преместване и други стандартни операции. Външният вид на прозорците и стандартните контроли по тяхната рамка зависят от настройките на

графичната среда на операционната система. Програмистът има само частичен контрол над външния вид на прозорците. Класът **Form** е наследник на класовете **Control**, **ScrollableControl** и **ContainerControl** и наследява от тях цялата им функционалност, всичките им свойства, събития и методи.

**CheckBox** е кутия за избор в стил "да/не". Свойството й **Checked** задава

дали е избрана.

**RadioButton** е контрола за алтернативен избор. Тя се използва в групи. Всички **RadioButton** контроли в даден контейнер (например форма) образуват една група и в нея само един **RadioButton** е избран в даден момент. **ListBox** контролата се използва за изобразяване на списък със символни низове, които потребителят може да избира чрез щракване с мишката

върху тях. По-важните свойства на тази контрола са:

- **Items** – колекция, която задава списъка от елементи, съдържащи се

в контролата.

- **SelectionMode** – разрешава/забранява избирането на няколко еле-

мента едновременно.

- **SelectedIndex**, **SelectedItem**, **SelectedIndices**, **SelectedItems** –

връщат избрания елемент (или избраните елементи).

**ComboBox** представлява кутия за редакция на текст с възможност за dropdown

алтернативен избор.

- **Text** – съдържа въведения текст.

- **Items** – задава възможните стойности, от които потребителят може

да избира.

- **DropDownStyle** – задава стила на контролата – дали само се избира

стойност от списъка или може да се въвежда ръчно и друга стойност.

**24. Повторно генериране на изключението. Изключения във вложени конструкции.**

Повторно генериране на изключе- ние

• Използва се, когато прихванатото изключение не може да бъде обработено;

• Синтаксис:

catch ( E x c e p t i o n T y p e p a r a m e t e r ) {

/ / . . .

throw;

}

• Повторно генериране на изключение може да се из- пълни само в рамките на catch-блок;

• Повторно генерираното изключение се обработва от следващият catch-блок;

Пример: Повторно генериране на из- ключение

|  |  |  |
| --- | --- | --- |
| 1 | #i n c l u d e | < i o s t r e a m > |
| 2 | #i n c l u d e | < e x c e p t i o n > |

3 using namespace s t d ;

4

5 void f u n ( void ) {

6 t r y {

7 c o u t < < " E x c e p t i o n t h r o w n i n f u n ( ) " < < e n d l ;

8 throw e x c e p t i o n ( ) ;

9 c o u t < < " T h i s s h o u l d n o t b e p r i n t e d " < < e n d l ;

|  |  |  |
| --- | --- | --- |
| 10 | } |  |
| 11 | catch ( e x c e p t i o n & | e x ) { |

12 c o u t < < " E x c e p t i o n h a n d l e d i n f u n ( ) " < < e n d l ;

|  |  |  |
| --- | --- | --- |
| 13 |  | throw; |
| 14 |  | } |
| 15 |  | c o u t < < " T h i s s h o u l d n o t b e p r i n t e d " < < e n d l ; |
| 16 | } |  |

**30. Същност на механизма на сериализация. Сериализиране на обекти с вградени класове**.

**Сериализация**

В съвременното програмиране често се налага да се съхрани състоянието

на даден обект от паметта и да се възстанови след известно време. Това

позволява обектите временно да се съхраняват на твърдия диск и да се

използват след време, както и да се пренасят по мрежата и да се възста-

новяват на отдалечена машина.

Проблемите при съхранението и възстановяването на обекти са много и за

справянето с тях има различни подходи. За да се намалят усилията на

разработчиците в .NET Framework е изградена технология за автомати-

зация на този процес, наречена **сериализация**. Нека се запознаем по-

подробно с нея.

**Какво е сериализация (serialization)?**

Сериализацията е процес, който преобразува обект или свързан граф от

обекти до поток от байтове, като запазва състоянието на неговите полета

и свойства. Потокът може да бъде двоичен (binary) или текстов (XML).

**Запазване на състоянието на обект**

Сериализацията се използва за съхранение на информация и запазване

на състоянието на обекти. Използвайки сериализация, дадена програма

може да съхрани състоянието си във файл, база данни или друг носител и

след време да го възстанови обратно.

можем да сериализираме обект

и да го запишем в бинарен файл със средствата на .NET Framework:

string str = ".NET Framework";

BinaryFormatter f = new BinaryFormatter();

using (Stream s = new FileStream("sample.bin", FileMode.Create))

{

f.Serialize(s, str);

}

При сериализирането на обекта в потока се записват името на класа, име-

то на асемблито (assembly) и друга информация за обекта, както и всички

член-променливи, които не са маркирани като **[NonSerialized]** (употре-

бата на този атрибут ще обясним по-нататък в тази тема). При десериали-

зацията информацията се чете от потока и се пресъздава обектът.

**Методи за сериализация**

**public static MemberInfo[] GetSerializableMembers(Type)**

Методът приема като параметър типа на класа, който ще бъде сериали-

зиран, и връща като резултат масив от **MemberInfo** обекти, съдържащи

информация за сериализируемите членове на класа.

**public static Object[] GetObjectData(Object, MemberInfo[])**

Методът приема като параметри обект, който ще бъде сериализиран и

масив с членовете, които трябва бъдат извлечени от обекта. За всеки от

тях се извлича стойността, асоциирана с него в сериализирания обект и

тези стойности се връщат като масив от обекти. Дължината му е същата,

като дължината на масива с членовете, извличани от обекта.

**59. Проблеми при присвояване и съвместимост на типовете.**

Макар С# да не инициализира автоматично локалните променливи, компилаторът предупреждава за неправилното им използване. Например следният код ще предизвика грешка при опит за компилация:   
  
  
  
Преобразуването на типове също е безопасно. CLR не позволява да се извърши невалидно преобразуване на типове – да се преобразува променлива от даден тип към променлива от тип, който не е съвместим с първия. При опит да бъде направено това, възниква изключение.   
  
Неявното преобразуване на типове е разрешено само за съвместими типове, когато не е възможна загуба на информация. При явно преобразуване на типове, ако те не са съвместими, се хвърля InvalidCastException по време на изпълнение. Например следният код предизвиква изключение по време на изпълнение:

**15. Свързване с база данни. Свързване на данни с контроли (Data Binding). DataGrid. Master-Details.Пример.**

База от данни се нарича всяка организирана колекция от данни.

Свързване на данни

Свързването на данни (data binding) осигурява автоматично прехвърляне на данни между контроли и източници на данни. Можем например да свържем масив, съдържащ имена на градове, с ComboBox контрола и имената от масива ще се показват в нея.Всички Windows Forms контроли поддържат свързване на данни (data binding). Можем да свържем което и да е свойство на контрола към източник на данни.

Контролата DataGrid

DataGrid контролата визуализира таблични данни. Тя осигурява навигация по редове и колони и позволява редактиране на данните. Като източник на данни най-често се използват ADO.NET DataSet и DataTable. Чрез свойството DataSource се задава източникът на данни, а чрез свойството

DataMember – пътят до данните в рамките на източника. По-важни

свойства на контролата са:

- ReadOnly – разрешава / забранява редакцията на данни.

- CaptionVisible – показва / скрива заглавието.

- ColumnHeadersVisible – показва / скрива заглавията на колоните.

- RowHeadersVisible – показва / скрива колоната в ляво от редовете.

- TableStyles – задава стилове за таблицата.

o MappingName – задава таблицата, за която се отнася дефинира-

ният стил.

o GridColumnStyles – задава форматиране на отделните колони –

заглавие, ширина и др.

Master-Details навигация

Навигацията "главен/подчинен" (master-details) отразява взаимоотношения от тип едно към много (например един регион има много области). В Windows Forms се поддържа навигация "главен/подчинен". За да илюстрираме работата с нея, нека разгледаме един пример: Имаме DataSet, съдържащ две таблици – едната съдържа имена на държави, а другата – имена на градове. Те са свързани помежду си така, че на всяка държава от първата таблица съответстват определени градове от втората

таблица:

Тогава можем да използваме две DataGrid контроли – първата, визуализираща държавите, а втората, визуализираща градовете, съответстващи на текущо избраната държава от първата контрола. За целта контролите се свързват с един и същ DataSet. На главната контрола се задава за източник на данни главната таблица. На подчинената контрола се задава за източник на данни релацията на таблицата:

// Bind the master grid to the master table

DataGridCountries.DataSource = datasetCountriesAndTowns;

DataGridCountries.DataMember = "Countries";

// Bind the detail grid to the relationship

DataGridTowns.DataSource = datasetCountriesAndTowns;

DataGridTowns.DataMember = "Countries.CountriesTowns";