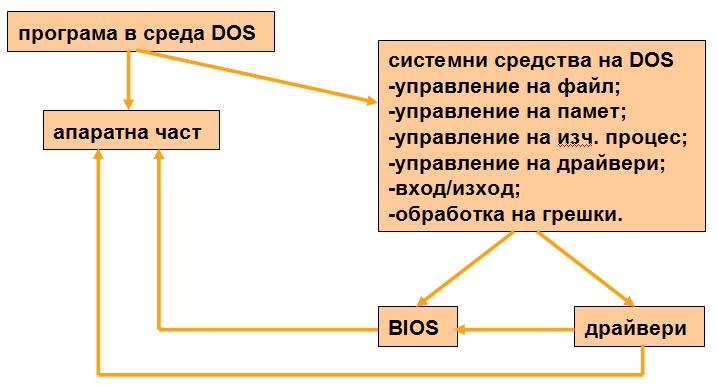
# 1. Въведение в програмирането в среда Windows. Вход чрез опашки и съобщения. Независим интерфейс. Многозадачност. Управление на паметта.

## 1. Въведение в програмирането в среда Windows.

****Системните средства се използват само ако програмата ги вика. Също се използва функционалност от драйвери и BIOS комуникацията може да се осъществи директно (директно свързване с апаратната част) или чрез посредничеството на драйвери (BIOS).

### UntitledПрограмна среда на Windows.

Windows компонентите използват драйвери или директна комуникация с апаратната част. Тези компоненти са:

***GDI* -** у-ства за графичната среда;

***Кernel*** - у-ства за изчислителния процес и паметта;

***User*-** у-ства за потребителски интерфейс и диалозите;

Всичко минава през Windows, който менажира задачите, заявките и т.н. Разликата между Windows и DOS е, че в Windows винаги се минава през Windows за да се получи достъп навсякъде.

### Особенности на ОС:

1. Вход чрез опашки и съобщения - отказваме се от DOS. Идеята е директно да извикаме ф-я или основно изчислителния процес да се изпълнява последователно. В DOS програмата лесно се проследява, но става по-трудно четима.

## Untitled2. В Windows се отказваме от директно извикване, последователн изпълнение на командите във файла.

Ако имаме приложение А.ЕХЕ, то стрелките показват от къде можем да получаваме съобщения- от самата себе си, от клавиатурата, от мишката, от софтуерни таймери и т.н. Съобщението е средство за насочване на изчислителния процес за изпълнение на определен програмен код, който е фрагмент от програмата. Приложението в Windows се състои от множество фрагменти, които се изпълняват ако са изпълнени определени събития (постъпили са определени съобщения). Съобщението е 1 число, което има буквен еквивалент в Windows. Има дефинирани около 150 съобщения които могат да се обработят или не. Всяко съобщение е съпътствано от параметри (поне 2). Msg-то идентифицира събитието, а параметрите дават допълнителна информация за msg-то. Ако msg-то е свързано с промяна размера на прозореца, то някъде ще се съдържа новата инф за размерите и след това ще се прерисува.

## 3. Независим интерфейс.

Изхода обикновено е на екран, но може да е на диск, интернет, принтер. Изхода трябва да е еднакъв, т.е. каквото се вижда на екрана това да излезе на принтера (примерно). Въпросът е да не се налагат промени. Ако се смени устройствения контекст (DC), в рамките на който се прави изхода, просто ще се насочи в друга посока. Мощен клас за управление в MFC е DC (Device Context). DC предоставя служебно създадена структура от данни, в която има набор от ф-ии чрез които могат да се настройват у-ва. Device Context трябва да се направи преди изхода, то тогава изхода се извежда към него.

## 4. Многозадачност.

В DOS няма многозадачност. Първите версии на Windows (3.0, 3.11, 95) не са многозадачни. Windows NT и 2000 са многозадачни. Характеристики на многозадачна ОС:

- Възможност за чист паралелизъм (ако хардуера го позволява) или превключване без намеса.

- Изолирани адресни пространства.

- Защити.

- Права на достъп.

- Приоритети.

- Преходи в състоянията на процесите (отложен, блокиран и т.н.).

*Правило*: изчислителният процес се управлява от поток приложения и всеки процес има собствено приложение => превключване на задачи става само ако се изчерпат msg-тата в опашката на първото, а се появят в опашката на другото msg.

## 5. Усъвършенстване управлението и извикването на файлове.

Има много механизми, които позволяват качване и сваляне на фрагменти от програмата, което позволява разширение на паметта (виртуалната памет, swap и т.н., асоциране на файл с ОП).

# 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

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

### Операционна система

Операционната система управлява ресурсите, процесите и потребителите на машината. Тя предоставя и някои услуги на приложенията като например: COM+, MSMQ, IIS, WMI и други.

Средата, която изпълнява .NET приложенията (CLR), е обикновен процес в операционната система и се управлява от нея, както останалите процеси.

### Common Language Runtime

Общата среда за изпълнение Common Language Runtime (CLR) управлява процеса на изпълнение на .NET код. Тя се грижи за заделяне и освобож-даване на паметта, управлява конкурентността, грижите за сигурността на приложенията и изпълнява други важни задачи, свързани с изпълнението на кода.

### Base Class Library

BCL представлява богата обектно-ориентирана библиотека с основни кла-сове, които осигуряват базова системна функционалност. BCL осигурява вход-изход, работа с колекции, символни низове, мрежови ресурси, сигурност, отдалечено извикване, многонишковост и др.

### ADO.NET и XML

Слоят на ADO.NET и XML предоставя удобен начин за работа с релационни и други бази от данни и средства за обработка на XML.

### Езици за програмиране

.NET Framework позволява на разработчика да използва различни езици за програмиране, както и да интегрира в едно приложение компоненти, разработвани на различни езици. Възможно е дори клас, написан на един език, да бъде наследен и разширен от клас, написан на друг език.

Microsoft .NET Framework поддържа стандартно езиците C#, VB.NET, Managed C++ и J#.Съвместимостта на езиците за програмиране в .NET Framework се дължи на архитектурни решения, които ще разгледаме в детайли след малко.

### Common Language Runtime

След като се запознахме накратко с архитектурата на .NET Framework, нека сега разгледаме в детайли и най-важният компонент от нея – CLR. Common Language Runtime (CLR) е сърцето на .NET Framework. Той пред-ставлява **среда за контролирано изпълнение на управляван код**. На практика CLR е тази част от .NET Framework, която изпълнява компили-раните .NET програми в специална изолирана среда.

В своята същност CLR представлява виртуална машина, която изпълнява инструкции, на езика IL (Intermediate Language), езикът до който се компилират всички .NET езици. CLR е нещо като виртуален компютър, който обаче не изпълнява асемблерен код за процесор Pentium, AMD или някакъв друг, а IL код.

### Управляван код

**Управляваният код (managed code)** е кодът, който се изпълнява от CLR. Той представлява поредица от IL инструкции, които се получават при компилацията на .NET езиците. По време на изпълнение управляваният код се компилира допълнително до машиннозависим код за текущата платформа и след това се изпълнява директно от процесора.

**Intermediate Language (IL)**

Междинният език Intermediate Language (IL), е език за програмиране от ниско ниво, подобен на асемблерните езици. За разлика от тях, обаче, IL е от много по-високо ниво, отколкото асемблерите за съвременните мик-ропроцесори. IL е обектно-ориентиран език. Той разполага с инструкции за заделяне на памет, за създаване на обект, за предизвикване и обработка на изклю-чения, за извикване на виртуални методи и други инструкции, свързани с обектно-ориентираното програмиране.

# 3. Движение на съобщенията в среда Windows. Примери на обработка на базови съобщения. Последователност на извличане на съобщенията от опашка.

- Липсва!!

# 4. Класическа структура на Windows приложение. Пример. Анализ на различията с конзолен еквивалента на същата програма. Стандартизирано именоване. Базови типове и структури.

**1. Класическа структура на Windows приложение.**

**Примернапрограма на DOS:**

*#include <stdio.h>*

*main()*

*{ printf (“Hello\n”); }*

Недостатъци:

1) Подразбира се изхода само на екран (не на принтер например)

2) Мястото на изведения текст не е фиксирано

3) Няма средства, които да гарантират еднотипното изпълнение при многократно стартиране

4) След края на програмата на екрана остава нещо

5) Не се поддържа мишка

***Приложение в Windows:***

1) Част от кода е стандартизиран и автоматично може да се генерира

2) В средата Windows има много подкрепи. Например ако се прави текстов редактор може да се постави 1 контрола.

*Пример на VC++:*

Switch (message)

{

case WM\_PAINT:

hdc=BeginPaint(hwnd, &ps);

GetClientRect(hwnd, &rect);

DrawText(hdc, “Hello”,-1, &rect,DT\_SINGLELINE | DT\_CENTER | DT\_VCENTER);

EndPaint(hwnd, &ps);

return 0;

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

}

return DefWindowProc(hwnd, message, wParam, lParam);

В примера се обхващат и обработват само 2 съобщения – WM\_PAINT и WM\_DESTROY, но има много други, които са генерирани. Тези, които не са прихванати се обработват по подразбиране

1 - отключване на заглавни файлове

2 - сигнатурата на WndProc

3 - WinMain стартова точка/подават й се параметри

4 - lpsz CmdParam параметри на командния ред

5 - дефинират се данни Handle към прозореца

***C#***

protected override OnPaint(PaintEventArgs pe)

{

Graphics g = pe.Graphics;

SolidBrush brush = new SolidBrush(Color.Red);

g.FillRectangle(brush, 0, 0, 10, 20);

base.OnPaint(pe);

}

private void Form1\_dbClick(EventArgs e)

{ this.BackColor =Color.White; }

**\* тип Msg**

wndclass.style = CS\_HREDRAW | CS\_VREDRAW Инициализираме полета на класа Style

RegisterClass(&wndclass); Регистрираме клас.

hwnd = CreateWindow (szAppName, “Name of the program” Създаване на прозорец (не визуализираме).

WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT,

CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, NULL, NULL, hInstance, NULL);

Показват размерите в началния момент. Те са по подразбиране => целия екран.

1ви-NULL - родителски прозорец

2ри-NULL - място hwnd към меню

3ти-NULL - стилове

# 5. Класическа структура на приложение в среда Windows: главна програма и proc- функция; прозоречен клас и регистрация на класа; създаване на прозорец и визуализация;структура на прозоречната функция.

Приложението трябва да има две обособени части

***1- WinMain*-** състои се от три части:

RegisterClass (регистрира класа),

CreateWindow (създава прозорец),

UpdateWindow(визуализира/ъпдейтва прозореца);

***2- WinProc* –** функция която съдържа множество swich-ове. Всяко едно приложение се състои тематично от три компонента – вход, данни и ресурси. Винаги кода е един, данните са собствени. Ресурсите са: менюта, икони, bitmap, др.

За създаване на Windows приложение е необходимо:

-1- редактор, вкойто се пише кода;

-2- компилатор, съдържащ нужните библиотеки;

-3- свързващ редактор за .obj файлове. Тези ресурси се прибавят към изходното .exe.

# 6. Обща класификация на класове в MFC среда. Структура на Windows приложение в MFC среда. Основни класове. Последователност при конструиране на класовете. Изчислителен процес.

## 1. 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..

## 2. Структура на Windows приложение с MFC подкрепа:

**Hello.h**

Class CMyApp: public SWinApp

{

public: virtual BOOL InitInstance();

};

class CMainWindow: public CFrameWnd

{

public: CMainWindow();

protected: afx\_msg void OnPaint();

DECLARE\_MESSAGE\_MAP()

};

**Hello.cpp**

#include <afxwin.h>

#include “Hello.h”

CMyApp myApp; //обект на прилож - той е само 1 в MFC прилож и има глобал. видимост

BOOL CMyApp::InitInstance() //създава обект от класа CMainWindow и копира неговия адрес в член-

{ променливата m\_pMainWnd

m\_pMainWnd = new CMainWindow;

m\_pMainWnd -> ShowWindow (m\_nCmdShow) //показва прозорецът

m\_pMainWnd -> UpdateWindow();

return TRUE;

}

BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)

ON\_WM\_PAINT()

END\_MESSAGE\_MAP()

CMainWindow::CMainWindow()

{

Create (NULL,\_T(“The Hello Application”));

}

**void** CMainWindow::OnPaint()

{ CPaintDC dc(this);

CRect rect;

GetClientRect(&rect);

dc.DrawText({T(“Hello”),-1,&rect, DT\_SINGLELINE\_CENTER); }

Към приложението:

\* ChelloApp – започва програмата.

\* AfxWinMain - първата използвана функция. Тя е скрита и не я пишем. Тази функция вика първата функция от нашето приложение ChelloWnd.

\* Всяко приложение се структурира в нов thread и няма previous instance. Тогава се стартира нов thread за приложението и се структурира нова опашка от която после се теглят съобщенията.

\*Следващото което се изпълнява - …. pMainWnd.

\*InitIntrance създава обекти от ChelloWnd като първо изпълнява конструктора m\_pMainWnd = new(HelloWnd), след което създава прозореца и после го визуализира. Тук нямаме Proc функция която да насочи изчисленията. Тук това става чрез картата на съобщенията. Чрез картата на съобщенията, възникващите съобщения се насочват към обработващите ги функции. Картата на съобщемията автоматично свързва съобщението с метода. В случая: WM\_PAINT с On\_Paint().

\* CframeWnd е наследник на FrameWnd

\* Create има около 10 параметъра аналози на параметрите в С++ (за Windows).

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

Картата на съобщението съпоставя съобщението и метод за обработката му. Подразбиращия се механизъм в една визуализационна среда е virtual method таблица за достъп до методите (чрез Runtime се конструира VMT). Чрез VMT би работил, но трябва да се направи VMT за всички съобщения, но може да се използва само едно. За това този механизъм е неефективен и е заменен с картата на съобщенията.

Картата на съобщенията се отнася в рамката на един клас, а не на цяло приложение. За всеки клас в който се генерира и обработва съобщението трябва да има карта на съобщенията. За да постави карта на съобщенията трябва:

1. В декларазията на класа да се постави DECLARE\_Massage\_Map.

2. В реализацията на класа трябва да се имплементира картата между Begin\_massage\_Map и

end\_massage\_map. Между тях се поставя запис за всяко интересуващо ни приложение.

В Begin\_massage\_Map има два параметъра. Първия е класа към който се структурира картата, а втория е неговия родител. Ако средата не предлага твърда врезка между съобщението и обработващата функция, то в картата трябва да се добави запис.

Тук (в нашия пример) е:

On\_MESSAGE(WM\_MYMESSAGE, OnMyMessage)

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

Декларацията на картата Declare трябва да е в раздела protected. Също така е хубаво да е накрая. Declare\_Message\_Map прави селедното:

1.Структурира масив от структури за всяко съобщение и съответстващата му функция.

2.Създава структура чиито полета сочат първия масив, а друг себеподобни структури в родителя.

Begin\_Message\_Map () - позволява да влезем в картата, т.е връща указател към картата.

Елементите на масива от структури създадени чрез Declare се попълват от елементите след BEGIN\_MESSAGE\_MAP.

Picture2 В програмата съобщението се обработва само в Hellownd (там само има Declaref\_Message\_Map). С Hellownd е наследник на CFrameWnd, която пък е наследник на CWnd. Ако примерно е дошло съобщение в C Hellownd-on.CREATE , то го няма и се търси обработка в родителския клас CFrameWnd, където има обработка и се обработва. Ако примерно дойде On Paint то се обработва директно в СHellownd , защото вече е декларирано.

*Забележка:*

1. Декларация TCHAR вместо char;

2. TCHAR\* или LPTSTR или LPTSTR вместо char\*;

3. дължината на знак е sizeof(TCHAR);

4. вместо стандартни run-time функции:strcopy()->tcscpy() или tcscall();

Кодови таблици и макрос \_Т

***“Hello” -ANSI***

L”Hello” -Unicode

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

Същото приложение във визуална среда

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

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

ASSERT\_VALID(pDoc);

// добавен код

CRect rcClient;

GetClientRext(rcClient);

pDC->DrawText

GetClientRect( rcClient );

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

}

# 8. Методика на изграждане на работоспособно приложение във визуална среда. Пример с VISUAL C++ приложение.

Picture1Едно 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 );*

*}*

# 9. Преход към програмиране за .NET - конзолно приложение. Програмиране за .NET с Windows форми. .NET версия на приложение: с диалог - без форма. .NET приложение с форма.

## 1. Кознолно приложение

**using** System;

class HelloCSharp

{

static void Main()

{

Console.WriteLine("Hello, C#");

}

}

- продуцира се малъкпо обем .exe файл на MS Intermediate Language (MSIL)

- след стартиране, .NET средата компилира междинния код към собствен машинен език за конкретния процесор и свързва с подходящите .DLL

-създал се е управляван код (managed code). Това е код, който може да се анализира

от друга програма по време на изпълнение, за да се определя обхвата на действията в него. Това особено личи при управление на паметта и в Internet среда.

## 2. WinForms вариант

**using** System;

using System.Windows.Forms;

public class SampleForm : System.Windows.Forms.Form

{

static void Main()

{

SampleForm sampleForm = new SampleForm();

sampleForm.Text = "Sample Form";

Button button = new Button();

button.Text = "Close";

button.Click +=

new EventHandler(sampleForm.button\_Click);

sampleForm.Controls.Add(button);

sampleForm.ShowDialog();

sampleForm.Dispose();

}

private void button\_Click(object sender, EventArgs e)

{

Close();

}

}

В него се създава прозорец, който съдържа бутон с текст "Close". При натискане на бутона прозорецът се затваря (това се реализира чрез прихващане и обработка на събитието "натискане на бутона").

- Дефиниран е клас SampleForm, който наследява класа System. Windows.Forms.Form. Този клас представлява главната форма на приложението.

- В главния метод Main() първо се задава заглавие за формата. След това се създава бутон, който се добавя в списъка с контролите на формата и се прихваща събитието "щракване върху бутона". Накрая формата се показва в модален режим (модален режим означава, че другите форми на приложението не са активни, докато не се затвори текущата) и след затварянето й се унищожава.

- При натискане на бутона се извиква събитие, което затваря формата, и приложението завършва.

**namespace** WindowsFormsApplication

{

static class Program

{

[STAThread]

static void Main()

{

string message = @"Това е само пример за

изпита по Програмни Среди";

string copyright = "\xA9 2009 Програмни Среди";

System.Windows.Forms.MessageBox.Show(message, copyright);

}

}

}

## Kаква е разликата между конзолно и Windows приложение:

1. Начина на комилация: /target :exe и /target :/winexe за отделна компилация и Console Application или Windows Application за Visual Studio.NET среда.

2. Конзолно приложение, стартирано през Windows създава прозорец за конзолен изход Windows Application не създава конзолен прозорец и всеки изход към конзола отива в небитието.

3. Не е нужна .NET CLR среда (поне System, System.Drawing, System.Windows.Forms).

4. Конзолно и windows приложения могат свободно да се смесват.

5. Реалната разлика е начинът по който програмата получава потребителски вход.

# 10. Добавяне на манипулатор на събитие. Структура с наследяване на форма. Манипулатори на събития или On… методи? Последователност на обработките.

Не се създава форма, а се наследява в наша форма

using System;

using System.Drawing;

using System.Windows.Forms;

class HelloWorld: Form

{

public static void Main()

{

Application.Run(new HelloWorld());}

public HelloWorld()//конструктор по подразбиране

{Text = “Hello World”;

BackColor = Color.White;

}

protected override void OnPaint(PaintEventArgs pea)

{

Graphics grfx = pea.Graphics;

grfx.DrawString(“Hello Windows Forms!”, Font, Brushes, Black, 0,0);

}

Последователността при наследяване от Control е следната:

1. Когато се вика On...() метод в наследяващ клас, то всъщност са предефинирани всички

On...() методи на родителите и кода в него ще бъде изпълнен

2. препоръчва се On...() методите в наследил клас да викат base.OnPaint(). Tака ще се достигне до On...() метода със същото име в класа Control. Понякога това не се прави.

3. Методът On...() на Control претърсва за всички инсталирани манипулатори за това събитие и последователно ги вика.

4. След като всички инсталирани манипулатори за събитието са извикани,

On...() от Control връща управлението на On...() метода в текущия клас

5. On() метода на текущия клас се изпълнява

**Наследяване на форми и контроли**

Windows Forms е проектирана така, че да позволява лесно наследяване и разширяване на форми и контроли. Това дава възможност за преизползване на общите части на потребителския интерфейс.

**Наследяване на форми**

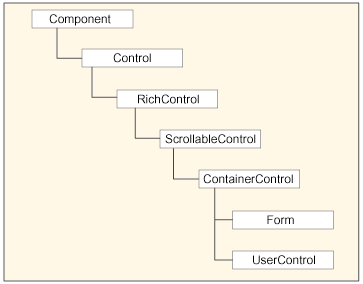
Наследяването на форми позволява повторно използване на части от потребителския интерфейс. Чрез него е възможно да променим наведнъж общите части на много форми. За целта дефинираме една базова форма, която съдържа общата за всички наследници функционалност.

Базовата форма е най-обикновена форма. Единствената особеност е, че контролите, които могат да се променят от наследниците, се обявяват като **protected**. При наследяване на форма се наследява класът на базовата форма.Не всички класове от Windows Forms са контроли. Някои са обикновени .NET компоненти, например **Menu**, **Timer** и **ImageList**. Изглежда малко странно защо менюто не е контрола, но това е така, защото компонентата **Menu** реално няма графичен образ и представлява списък от **MenuItem** елементи. **MenuItem** класът вече има графичен образ и следователно е контрола.

# 11. Windows Forms в .NET. Контроли и йерархия на графичните контролите. 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:

Прозорец, к. има специфика в обработването, предопределена от типа. Най-често предава msg към родителя си. Най-честото съобщ. е WM\_COMMAND, а най простият контрол е BUTTON.

определяне на родител: hwndParent = GetParent(hwndChild);

съобщения към родител: SendMessage(hwndParent, message, wParam, lParam);

# 12. Диалози – стандартни и потребителски. Видове и приложение. Пример за употреба.

При разработката на Windows Forms приложения често пъти се налага да извеждаме диалогови кутии с някакви съобщения или с някакъв въпрос. Нека разгледаме стандартните средства за такива ситуации.

Стандартни диалогови кутии Класът MessageBox ни позволява да извеждаме стандартни диалогови кутии, съдържащи текст, бутони и икони:

- съобщения към потребителя

- въпросителни диалози

Показването на диалогова кутия се извършва чрез извикване на статичния метод Show(…) на класа MessageBox.

Следният код, например, ще покаже диалогова кутия със заглавие "Предупреждение" и текст "Няма връзка с интернет":

MessageBox.Show("Няма връзка с Интернет.", "Предупреждение");

Пример за стандартна диалогова кутия с малко повече функционалност:

bool confirmed =MessageBox.Show("Наистина ли ще изтриете това?", "Въпрос", MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes;

Този код ще покаже диалогова кутия със заглавие "Въпрос" и текст "Наистина ли ще изтриете това?". Преди текста ще има икона с въпросителен знак в нея, а под него – бутони Yes и No. Ако потребителят натисне Yes, променливата confirmed ще има стойност true, в противен случай ще има стойност false.

Извикване на диалогови кутии освен стандартните диалогови кутии можем да използваме и потребителски дефинирани диалогови кутии. Те представляват обикновени форми и се извикват модално по следния начин:

DialogResult result = dialog.ShowDialog();

Методът ShowDialog() показва формата като модална диалогова кутия. Типът DialogResult съдържа резултата (OK, Yes, No, Cancel и др.) от извикването на диалога. Задаването на DialogResult може да става автоматично, чрез свойството DialogResult на бутоните, или ръчно – преди затварянето на диалога чрез свойството му DialogResult.

# 13. SDI и MDI приложения. Структура и пример.

MFC прави лесно да се работи едновременно с един документен интерфейсни (SDI) и няколко документни интерфейсни (MDI) приложения.  
SDI приложенията позволяват само един отворен прозорец на документ кадър по кадър. MDI приложенията позволяват няколко документа да бъде отворени в една и съща инстанция на приложението. Приложението MDI има прозорец, в който няколко MDI прозорци, които са прозорци сами по себе си, могат да бъдат отворени, всяка от които съдържа отделен документ. В някои приложения, детето прозорци могат да са от различен тип, като например прозорците на графиките и таблиците прозорци. В този случай, на лентата с менюта може да се променят MDI прозорците на различни видове ,които са активирани.

Осн. развичия в SDI и MDI прилож. Много изгледи в MDIпроблемът за синхронизация на промените в изглед.

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.

В SDI има един обект на приложението на документа. Това означава , че за да отвори нов документ трябва да затворим стария.

В MDI имаме много документални обекти, затова не е необходимо припокриване. При избиране на нов се създава нов обект.Ако е ….. Template то се взема от него документен шаблон и се създава , а ако са повече то се извежда та екрана от кой точно шаблон да се вземе.

Main Frame Windows

**f-- 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**

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

## 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.

MappingName – задава таблицата, за която се отнася дефинира нейният стил.

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";

## DataBinding fundamentials

Основен е BindingSource компонета – за свързване към източници на данни с богати възможности за управление. Добавянето става в развойната среда : Data Source Configuration Wizard (Data| Add New Data Source). Източник на данни може да е : база данни, Web услуга, обект;

При източник БД – добавя DataSet обект. Позволява редактиране, преконфигуриране или refresh над DataSet. Контроли като DataGridView когато се вкарат (droped) във форма, автоматично питат за data source.

## BindingSource

Свързване с източник на данни обикновено става през DataSet (за БД от релационен тип);

Свързване може да има и с : таблици, XML данни, .NET обекти. Всички те могат да се привързват като източници към форма. Това става с определени усилия: напр реализация на IBindingList интерфейса.

Компонентът BindingSource автоматизира това за всеки тип източник на данни и имплементира горния интерфейс вместо програмиста.

За да се конфигурира свързване към някакъв тип източник на данни:

1. drop на DataSource компонент във формата

2. подходяща настройка на properties: DataSource и DataMember. Ето пример за свързване с DataSet:

**Public** class DataBindingForm : Form {

**Public** DataBindingForm() {

……

this.BindngSource.DataSource = this.northwindDataSet;

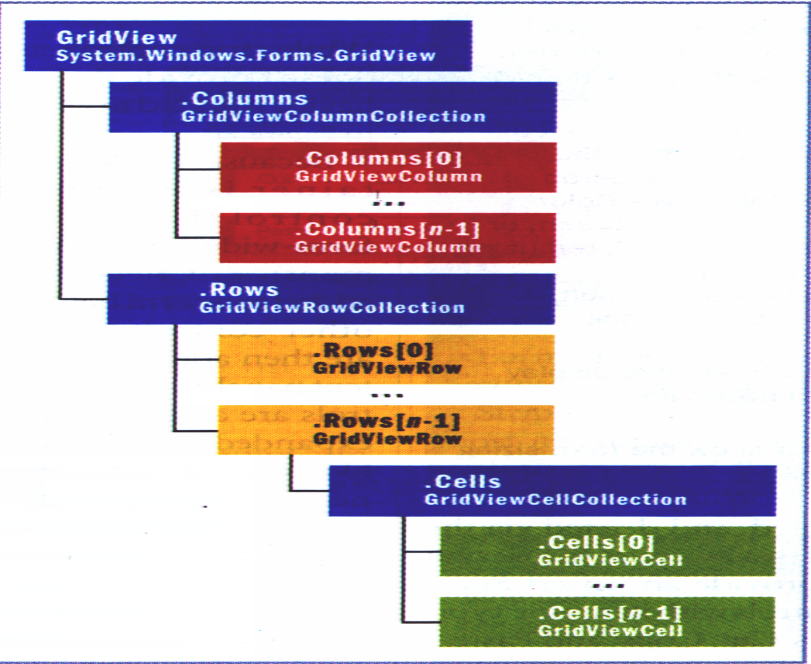
this.bindingSource.DataMember = “Employees”; }}

Оттук нататък източника на данни за тази форма и контролите в нея е дефиниран

**DataGridView** – обогатен grid контрол

(старият DataGrid още съществува – за навигация и редактиране на йерархични данни)

Новост е обогатеният обектен модел на DataGridView:



**DataGridView** позволява:

Лесна настройка на стилове, формати;

Изобразяване на различни типове данни, вкл. Картинки;

Мощни възможности за динамично преконфиггуриране (напр. Пренареждане на колони).

Повече от 100 привързани с него events за navigation, editing, validation, error handling и т.н.

Запълване на DataGridView с данни обикновено става след привързване към data source ( с настройка на property DataSource:

# 15. GDI+. Методи за изчертаване на основните фигури. Запълване, контур и специални ефекти.

Пакетът System.Drawing осигурява достъп до GDI+ функциите на Windows:

- повърхности за чертане

- работа с графика и графични трансформации

- изчертаване на геометрични фигури

- работа с изображения

- работа с текст и шрифтове

- печатане на принтер

Той се състои от няколко пространства:

- **System.Drawing** – предоставя основни класове като повърхности, моливи, четки, класове за изобразяване на текст.

- **System.Drawing.Imaging** – предоставя класове за работа с изображения, картинки и икони, класове за записване в различни файлови формати и за преоразмеряване на изображения.

- **System.Drawing.Drawing2D** – предоставя класове за графични трансформации – бленди, матрици и др.

- **System.Drawing.Text** – предоставя класове за достъп до шрифтовете на графичната среда.

- **System.Drawing.Printing** – предоставя класове за печатане на принтер и системни диалогови кутии за печатане.

## Класът Graphics

Класът System.Drawing.Graphics предоставя абстрактна повърхност за чертане. Такава повърхност може да бъде както част от контрола на екрана, така и част от страница на принтер или друго устройство. Най-често чертането се извършва в обработчика на събитието Paint. В него при необходимост се преизчертава графичния облик на контролата. Параметърът PaintEventArgs, който се подава, съдържа Graphics обекта. Graphics обект може да се създава чрез Control.CreateGraphics(). Той задължително трябва да се освобождава чрез finally блок или с конструкцията using, защото е ценен ресурс. Чрез примера ще илюстрираме работата с GDI+ чрез пакета System.Drawing – чертане на геометрични фигури с четки и моливи и изобразяване на текст със зададен шрифт.

private void MainForm\_Paint(object sender,

System.Windows.Forms.PaintEventArgs e)

{

Graphics g = e.Graphics;

g.SmoothingMode = SmoothingMode.AntiAlias;

Brush brush = new SolidBrush(Color.Blue);

g.FillEllipse(brush, 50, 40, 350, 250);

brush.Dispose();

Pen pen = new Pen(Color.Red, 2);

g.DrawRectangle(pen, 40, 50, 200, 40);

pen.Dispose();

brush = new SolidBrush(Color.Yellow);

Font font = new Font("Arial", 14, FontStyle.Bold);

g.DrawString(".NET Framework", font, brush, 60, 60);

brush.Dispose();

font.Dispose();

}

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 );

}

# 16. Вход/Изход в .NET. Работа с файлове, директории, потоци, четци и писци.

## Основни видове и особености

Потоците в обектно-ориентираното програмиране са една абстракция, с която се осъществява вход и изход от дадена програма. Потоците в C# като концепция са аналогични на потоците в други обектно-ориентирани езици, напр. Java, C++ и Delphi (Object Pascal).

**Потокът** е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата с устройство за съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоставят средства за четене и запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни операции в .NET Framework.Потоците в .NET Framework се делят на две групи – **базови** и **преходни**. И едните, и другите, наследяват абстрактния клас **System.IO.Stream**, базов за всички потоци.Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранение, като файловата система (например класът **FileStream**), паметта (**MemoryStream**) или данни, достъпни по мрежата (**NetworkStream**). По-нататък ще разгледаме класа **FileStream** в точката "Файлови потоци". Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като при това посредничество добавят допълнителна функци-оналност, например буфериране (**BufferedStream**) или кодиране (**CryptoStream**). По-подробно ще разгледаме **BufferedStream** в точката "Буферирани потоци".За четене на данни от поток се използва методът **int Read(byte[] buffer, int offset, int count)**. Той чете най-много **count** на брой байта от текущата позиция на входния поток, увеличава позицията и връ-ща броя прочетени байтове или **0** при достигне края на потока. Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода **NetworkStream.Read(…)**, а не са налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **NetworkStream. DataAvailable**, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операция **Read()** ще блокира или ще върне резултат веднага.

## Писане в поток

Методът **Write(byte[] buffer, int offset, int count)** записва в изход-ния поток **count** байта, като започва от зададеното отместване в байтовия масив. И тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записани в потока с **Write(…)**, са достигнали до местоназначението си след успеш-ното изпълнение на метода. Възможно е потокът да буферира данните и да не ги изпраща веднага.Файловите потоци в .NET Framework са реализирани в класа **FileStream**, който вече беше използван в примера за потоци. Като наследник на **Stream**, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.Четенето и писането във файлови потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа **Stream** – с методите **Read()**, **Write()** и т. н.

Файловите потоци поддържат пряк достъп до определена позиция от файла чрез метода **Seek(…)**.

## Четците и писачите

Readers and writers в .NET Framework са класове, които улесняват работата с потоците. При работа например само с файлов поток, програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени четенето и записа на различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват двоични и текстови. Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar()**, **ReadChars()**, **ReadInt32()**, **ReadDouble()** и др. за четене и съответно **Write(char)**, **Write(char[])**, **Write(Int32)**, **Write(double)** – за запис. Може да се чете и записва и **string**, като той се представя във вид на масив от символи и префиксно се записва дължината му – **ReadString()**, респ. **Write(string)**.Текстовите четци и писачи осигуряват четене и запис на текстова инфор-мация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактните класове **TextReader** и **TextWriter**. Основните методи за четене и запис са следните: - **ReadLine()** – прочита един ред текст.

- **ReadToEnd()** – прочита всичко от текущата позиция до края на потока.

- **Write(…)** – вмъква данни в потока на текущата позиция.

## Работа с директории. Класове Directory и DirectoryInfo

Класовете **Directory** и **DirectoryInfo** са помощни класове за работа с директории. Ще изброим основните им методи, като отбележим, че за **Directory** те са статични, а за **DirectoryInfo** – достъпни чрез инстанция.

- **Create()**, **CreateSubdirectory()** – създава директория или подди-ректория.

- **GetFiles(…)** – връща всички файлове в директорията.

- **GetDirectories(…)** – връща всички поддиректории на директорията.

- **MoveTo(…)** – премества (преименува) директория.

- **Delete()** – изтрива директория.

- **Exists()** – проверява директория дали съществува.

- **Parent** – връща горната директория.

- **FullName** – пълно име на директорията.

# 17. Windows Presentation Foundation (WPF).XAML. Контроли и логическо дърво. Примери.

Нова рендираща система, базирана на DirectX

–Осигурява поддръжка на хардуерно ускорение

–Поддръжка на ефекти

–Вградена поддръжка на 3D

•Добра интеграция на 2Dи 3D UI

Независим от резолюцията!

•Декларативно програмиране –XAML

•Добри инструменти за разрабтване на GUI –Blend

•Стилове и теми

•Вградени анимации

•Kомпозиране на елементи

•Разделяне на данните(Data)от поведението (Behavior)

•Лесно разпространение

–ClickOnce

–Browser(XBAP)

## XAML

XML базиран език=>тагове и атрибути

•Декларативен

–Разделение на описание от поведение

•Описва .NETобекти

•Използва се за описване на потребителски интерфейс–работи с класовете от WPFплатформата

## Примери

Пример1:

<Button>

<StackPanel Orientation=”Horizontal”>

<Image Source=”speaker.png” Stretch=”Uniform”/>

<TextBlock Text=”play sound”/>

<StackPanel/>

</Button>

Пример2:ComboBox with live preview

<Window xmlus = “http://schemas.microsoft.com/winfx/2000/xaml/presentation”

Xmlus:x = “http://schemas.microsoft.com/winfx/2006/xaml”

Xmlus:l = “clr-namespace : LivePreviewComboBox”>

<StackPanel>

<TextBlock Text = “Preview Value:” />

<TextBlock Text = “Binding LivePreview Item, Element Name = liveBox:” />

<L:LivePreviewComboBox x:Name = “LiveBox” />

</StackPanel>

</Window>

Как да създадем бутон:

<!--XAML-->

<Button Content="OK"/>

//C#

Button b= new Button() {Content = "Ok“ };

Еквивалентно

Таг –класа на обекта

Атрибут –променя стойност на свойство

Property елементи:

Не създават нови обекти

•Присвоява стойност на свойство

<Button>

<Button.Content>

<RectangleHeight="40"

Width="40"

Fill="Black" />

</Button.Content>

</Button>Таг –класа, собственик на свойството и името на свойството

## Основни класове на WPF:

•DispatcherObject

•DependencyObject

•Visual

•UIElement

•Framework Element

•Control

•Shapesи Text, ContentPresenter

•Control, ContentControl, UserControl

•Window

## Контроли на WPF:

Content Controls

–Buttons: Button, RepeatButton, ToggleButton, CheckBox, RadioButton

Items Controls

–ItemsControl

–ListBox

–ListView

–ComboBox

–Menus: Menu, ContextMenu

Програмиране с .NETиWPF31

–TreeView

–ToolBar

–StatusBar

# 18. LINQ – Language INtegrated Query. Query Expressions – Заявки вградени в езика. Ламбда 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””

# 19. Таймер (Timer). Работа с таймери. Пропъртита, събития. Пример.

## Пример

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 }

Често в приложенията, които разработваме, възниква необходимост от изпълняване на задачи през регулярни времеви интервали. Таймерите предоставят такава услуга. Те са обекти, които известяват приложението при изтичане на предварително зададен интервал от време. Таймерите са полезни в редица сценарии, например, когато искаме да обновяваме периодично потребителския интерфейс с актуална информация за статуса на някаква задача или да проверяваме състоянието на променящи се данни.

## System.Timers.Timer

Класът предоставя събитие за изтичане на времевия интервал **Elapsed**, което е делегат от тип **ElapsedEventHandler**, дефиниран като:

public delegate void ElapsedEventHandler(object sender, ElapsedEventArgs e);

При изтичане на интервала, указан в свойството **Interval**, таймерът от тип **System.Timers.Timer** ще извика записалите се за събитието методи, използвайки нишка от пула. Ако използваме един и същ метод за получаване на събития от няколко таймера, чрез аргумента **sender** можем да ги разграничим. Класът **ElapsedEventArgs** чрез свойството **DateTime SignalTime** ни предоставя точното време, когато е бил извикван метода. За стартиране и спиране на известяването, можем да извикаме съответно **Start()** и **Stop()** методите. Свойството **Enabled** ни позволява да инструктираме таймера да игнорира събитието **Elapsed**. Това прави **Enabled** функционално еквивалентно на съответните **Start()** и **Stop()** методи. Когато приключим с таймера, трябва да извикаме **Close()**, за да освободим съответните системни ресурси.

## System.Threading.Timer

**System.Threading.Timer** прилича на **System.Timers.Timer** и също използва пула с нишки. Основната разлика е, че той позволява малко по-разширен контрол – може да указваме кога таймера да започне да отброява, както и да предаваме всякаква информация на метода за обратни извиквания чрез обект от произволен тип. За да ползваме **System.Threading.Timer**, трябва в конструктора му да подадем делегат от тип **TimerCallback**, дефиниран като:

public delegate void TimerCallback(object state);

При всяко изтичане на времевия интервал, ще бъдат извиквани методите в този делегат. Обикновено като обект за състояние има полза да подаваме създателя на таймера, за да можем да използваме същия метод за обратни извиквания за обработка на събития от множество таймери. Другият параметър в конструктора на таймера е времевият интервал. Той може и да бъде променен впоследствие с извикване на **Change(…)** метода.

**System.Threading.Timer** не предлага удобен начин за стартиране и спиране. Неговата работа започва веднага след конструирането му (по-точно след изтичането на подаденото стартово време) и прекъсването му става само чрез **Dispose()**. Ако искаме да го рестартираме трябва да създадем нов обект.

## System.Windows.Forms.Timer

Пространството от имена **System.Windows.Forms** съдържа още един клас за таймер, който е със следната дефиниция:

public class Timer : Component, IComponent, Idisposable

{

public Timer();

public bool Enabled{virtual get ; virtual set;}

public int Interval {get; set;}

public event EventHandler Tick;

public void Start();

public void Stop();

}

Въпреки, че методите на **System.Windows.Forms.Timer** много приличат на тези на **System.Timers.Timer**, то **System.Windows.Forms.Timer** не използва пула с нишки за обратните извиквания към Windows Forms приложението. Вместо това, през определено време той пуска Windows съобщението **WM\_TIMER** в опашката за съобщения на текущата нишка. Използването на **System.Windows.Forms.Timer** се различава от употребата на **System.Timers.Timer**, само по сигнатурата на делегата за обратни извиквания, който в случая е стандартният **EventHandler**.

# 20. MVC (Model-View-Controller) във Windows Forms.Пример.

# 21. Езикови средства за създаване устойчив код. Предоставени от 32 битова среда възможности за целта. Въведение в SEH механизма. Терминираща обработка (\_\_finally). Възможни двусмислия .

## 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.

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

## 32 битова сред

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 той 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.

## 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.

\_\_leave – поставена в try води до преждевременно пренасочване към блок finally като нищо от това което е предвидено за изпълнение след leave в try няма значение, както и междинното съхранение

# 22. Филтър за обработка на изключения (\_\_except). Събиране информация 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 изкл. събития (C++ Exception Handling). Стандартни обекти в ANSI C++ стандарт. Обработка на изключения при конструиране на обект.

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.

2. Standart Exception Library

***exeption***

***logic\_error runtime\_error***

***domain\_error range\_error***

***(вътр. грешка) overflow er.***

***invalid\_argument***

***length\_error***

***out\_of\_range***

- runtime error - групира всички проблеми, които не могат да се групират при изпълнението

- logic error - групира всички проблеми, които се пораждат при изпълнение на ф-ята.

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; }}

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

• 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

#include < i o s t r e a m >

#include < e x c e p t i o n >

using namespace std ;

void fun(void)

{

try {

cout < < " Exception thrown in fun()" < < e n d l ;

throw exception() ;

cout<<"This should not be printed" < < e n d l ;

}

catch ( e x c e p t i o n & e x ) {

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 ;

throw;

}

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 ;

}

# 25. Съвместяане на механизмите за реакция на изкл. събития (Win32 SEH и C++ Exception Handling).

## Съвместяване на механизмите за реакция на изключителни събития.

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 преобразуването 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);

}

# 26.Изключения в .NET . Дефиниране на собствено изключение.

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.

Правила за работа с изключения

1. разработвате библиотека: ако прихванете всички изкл. , как разработващия приложение с библиотеката ще знае че нещо се е случило

2. разработвате библиотека с типове – не винаги знаете кое е грешка, кое не. Оставете това на викащия

3. Избягвайте код , прихващащ всичко: catch(System.Exception) {………}

4. Ако операция е частично завършена изключение и следва възстановяване в начално съст. Най-добре прихванете всичко, възстановете и уведомете (с друго изкл. ) викащата страна.

5. След прихващане и обработка на изключение, често е добре да уведомите извикващия: подавате същото (само с throw) или друго изключение (това е начина за преобразуваме изключението от нещо специфично, към общоразбираемо за потребител).

Необработвани съобщения (такива, които никой catch не разпознава)

Най напред следва да се разработи единна политика за тях – напр. въведен текст се съхранява и се визуализира диалогов прозорец с информация и т.н.;

1.При отдалечено викана процедура или web услуга или сървърно-базиран код, който подава exception, то той се изпълнява в сървърно обкръжение на try/catch. Тъи като exception обекта е сериализиран, той може да се предава през граница на Domain – т.е. обратно към клиентското приложение.

2. В общия случай, необработени съобщения могат да се насочват за обработка към дефинирана в в приложението делегатна функция, регистрирана като

event handle от тип System.UnhandledExceptionEventHandler към стандартния тип за изключния: System.AppDomain.UnhandledException . Пример:

AppDomain.CurentDomain.UnhandledException +=

new UnhandledExceptionEventHandler(MyUnhandledExceptionFunction);

3. Необработваните изключения в приложения, базирани на Windows Forms се прихващат така: цялата WinProc ф-ия всъщност, се вика в обхващащ я автоматично try/catch.

При наличие на необработено по-долу изключение, catch блокът извиква виртуалния метод OnThreadException() дефиниран в System.Windows.Forms.Control и предефиниран в Application

Той визуализира стандартен прозорец за ‘unhandled exception’

Можете да предефинирате поведението чрез ваш метод от делегатен тип

System.Threading.ThreadExceptionEventHandler

и след това да свържете този метод с ThreadException събитието на класа Application

4. Необработени съобщения в ASP.NET

ASP обхваща кода на приложението в собствен try блок и предопределя начин за обработка. Може да се намесите като регистрирате свой callback метод към събитие Error на класа System.Web.UI.Page или на клас System.Web.UI.UserControl

(методът може и да се вика за всяко необработено изключение от която и да е страница на

приложението – ако callback метода е свързан с Error събитие на клас

System.Web.HTTPApplication)

5. Необработени изключения в среда ASP.NET XML

Отново обхващащ кода try блок на ASP.NET подава SoapException обект. Той се сериализира в XML вид и може да се предава към друг компютър или приложение, работещо като клиент на XML Web услугата.

# 28. Вход/изход и сериализация. Работа с файлови обекти без сериализация. Цел и предимства на архивните 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();}

CFile – капсулира всички AVI ф-ии. Има над 25 м-да-за всички възможни метода над файла.

- отваряне на файл – има предвид “е” клас. Декл. ме обект от CFile exception. Отваряне на файл в режим за четене и запис. Може с try и catch. При отваряне на файл се показва режима.

**-** затваряне:

А. file.Close();

В. 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

**М**етодите за работа с CFile класове могат да се използват и в др. случаи. Напр. MemFile и SharedFile, DataObject – пренасянето на данни м/у приложения, може да се стандартизира чрез него, а за програмата си е все същото. SocketFile – предоставя файл с подобен интерфейс. Stdio File – специален файл към текстови данни.

Internet File – все едно се работи с файл, но на практика се работи с протоколите от нета.

**CMemFile, CSharedFile**

COleDataObject::GetFileData

CSocketFile, стои между архивния обект и CSocket обекта.

CStdioFile , наследник на CFile.

**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, независимо от средата. За да се осъществи това посредата ! да се постави нещо. Архива е винаги между нашето приложение 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.

# 29. Създаване на собствени класове, поддържащи сериализация. Пряко използване на Serialize().

Голямо предимство на сериализацията в MFC среда е, че можем да създаваме собствени, при това сериализирани класове, без да предефинираме команди - Ако създаваме собствени класове, и ако искаме те да поддържат променливи от този клас и ако направим този клас наследник на CObject, тогава не е неоходимо да предефинираме командите, компилатора автоматично добавя код за това.

**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**

# 30. Версии при сериализации. Работа с указатели към обекти и проблеми при използване на псевдоними на обекти при сериализация.

Ако след време искаме да променим нашия клас, тогава трябва да преопишем Serialize, но пък написните преди това програми трябва да се информират, че версията е различна. Това става като се сравнят номерата на проектите: IMPLEMENT\_SERIAL(cLine, CObject, 1| VERSIONABLE\_SHEMA)

Ако открием разлика тогава се генерира exeption. Това е стандартен механизъм. Но ако искаме да подтиснем се използва VERSIONABLE\_SHEMA и се приема че може да се работи с различни версии.

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

Когато се обръщаме директно към Serialize се отказваме от проверка за проекти (само една версия). Операторите за сериализация преобладават CSrting в UniCode.

# 31. Същност на механизма на сериализация. Сериализиране на обекти с вградени класове.

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.

# 32. Run-time сериализация в .NET.

System.Runtime.Serialization – обединява типове, отговорни за процеса на сериализация и десериализация на обекти (запазване на състоянието на обект и по-късното му възстановяване).

.NET Framework ни осигурява два стандартни форматера, дефинирани в пространство­то System.Runtime.Serialization:

* BinaryFormatter – сериализира обект в двоичен формат. По­лу­че­ният в резултат на сериализацията поток е много компактен.
* SoapFormatter – сериализира обект в SOAP формат. За разлика от двоичния формат, SOAP форматът осигурява съвместимост с други системи, защото представлява XML-базиран стандарт за обмяна на съобщения и е независим от платформата. SOAP стандартът ще разгледаме в детайли в темата за уеб услуги.

Можем да създаваме потребителски дефинирани форматери. Те на­следя­ват абстрактния клас Formatter, осигуряващ базова функционалност.

using System;

using System.Runtime.Serialization;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

public sealed class Program {

public static void Main() {

string data = "this must be stored in a file";

FileStream fs = new FileStream("SerializedString.Data",

FileMode.Create);

BinaryFormatter bf = new BinaryFormatter();

bf.Serialize(fs, data);

fs.Close();

}

}

# 33. 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

1. Когато имаме много instance на едно приложение всички използват общ код и ресурси, само данните са различни.
2. Елементите в паметта са преместваеми
3. Сегментите както и ресурсите най-често се зареждат при повикване.
4. Работи механизма за отстраняване на блоковете от паметта.

Имае 5 модела памет: small, medium, compact,large, huge

**Small** – с 1 кодов и 1 даннов сегмент- модел за малки програми

**Medium** – бързи програми с много кодови и един даннов сегмент,работи се с близки указатели към данни.

**Compact –** 1 кодов и много даннови сегменти, но той не е много добър

**Large** – модел с много кодови и с много даннови сегменти

**Huge** - много кодови и даннови сегменти, но работи с huge указтели. Това са далечни указатели след нормализация.

# 34. Работа с динамични блокове памет. Предимства на използването на собствени динамични блокове в рамките на процес, нишка или клас.

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 хийп:

- 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

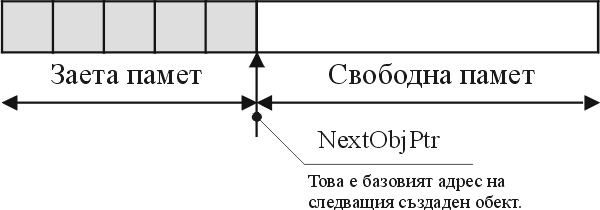
3) проверява ако е 0 унищожава handler към файла

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();

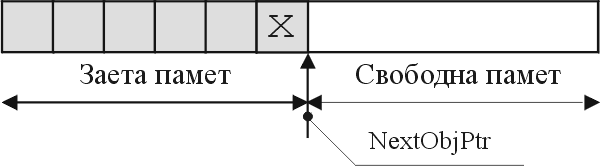
C# компилаторът превежда кода в IL newobj инструкция:

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.

# 35. Memory- mapped файлове. Избягване на фрагментацията при работа с малки блокове памет.

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.

## Съвети 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.

# 36. Стратегии на управление на памет и събиране на ‘боклук’ в .NET среда. Алгоритъм за “събиране на боклук”

## Освобождаване на неизползваните обекти

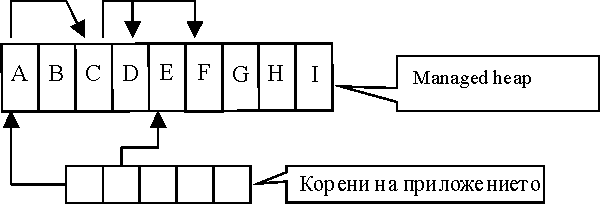
След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed heap има обекти, които вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключване на работата по събиране на отпадъци се възобновява работата на всички нишки и приложението продължава своето изпълнение. Както вероятно се досещате, откриването на ненужните обекти и освобождаването на ресурсите, заети от тях, не е проста задача. В тази секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване.

За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките на приложението в дадения момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване. Възниква въпросът как garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която системата за почистване на паметта започва своята работа.

## Корени на приложението

Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обекти от managed heap, или са установени на **null**. Например всички глобални и статични променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или параметри в стека към момента, в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съдържащи указатели към обекти, също са част от корените. Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ще стане дума в секцията за финализация на обекти в настоящата глава. Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на приложението). Когато JIT компилаторът компилира IL инструкциите на даден метод в процесорни инструкции, той също съставя и вътрешна таблица, съдържаща корените за съответния метод. Тази таблица е достъпна за garbage collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да определи кои са корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се определят корените зa всички извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и тези, намиращи се в глобални и статични променливи. Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъде считан за отпадък. JIT компилаторът може да определи кога този обект се достъпва от кода за последен път и веднага след това го изключва от вътрешната таблица на корените, с което той става кандидат за почистване от garbage collector. Изключение правят случаите, когато кодът е компилиран с **/debug** опция, която предотвратява почистването на обекти, които са в обхват. Това се прави за улеснение на процеса на дебъгване – все пак при трасиране на кода бихме искали да можем да следим състоянието на всички обекти, които са в обхват в дадения момент.

## 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:

# 37. 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.

## Недостатъци на Finalize():

- заделяне на финализирани обекти е по-бавно

- осводождаването се забавя, особено при масиви от такива обекти

- финализиран обект, рефериращ други обекти забавя финализирането им

- няма контрол кога се изпълнява Finalize()

- няма гаранция за реда на изпълнения на Finalize(). à във Finalize() да не се викат вътрешни обекти.

## Кога се вика Finalize():

- поколение 0 се е запълнило;

- явно обръщение към System.GC.Collect()

- процес завършва – вика се Finalize() за всички обекти в него.

## 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.

# 38 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.

# 39. 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:

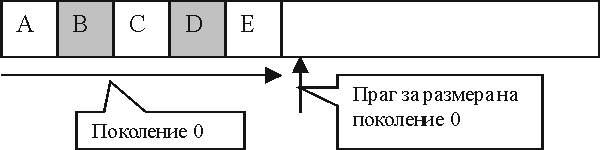
* колкото по-нов е един обект, толкова по-вероятно е животът му да е кратък. Типичен пример за такъв случай са локалните променливи, които се създават в тялото на даден метод и излизат от обхват при неговото напускане.
* колкото по-стар е обектът, толкова по-големи са очакванията той да живее дълго. 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.

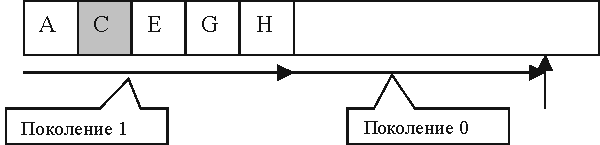
Нека сега предположим, че е минало още известно време, през което приложението е създавало обекти в динамичната памет. Managed heap сега изглежда по следния начин:



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.

# 40. Предварително оценяване за големи обеми памет.

# 41.NET Framework и системата за управление на общи типове. Типовете в CLR.

## Какво е CTS?

CLR поддържа много езици за програмиране. За да се осигури съвмес­тимост на данните между различните езици е разработена общата система от типове (Common Type System – CTS). CTS дефинира поддържа­ните от CLR типове данни и операциите над тях.

## CTS и езиците за програмиране в .NET

Всички .NET езици използват типовете от CTS. За всеки тип в даден .NET език има някакво съответствие в CTS, макар че понякога това съответ­ствие не е директно. Обратното не е вярно – съществуват CTS типове, които не се поддържат от някои .NET езици.

## CTS e обектно-ориентирана

По идея всички езици в .NET Framework са обектно-ориентирани. Common Type System също се придържа към идеите на обектно-ориентираното програмиране (ООП) и по тази причина описва освен стандартните типове (числа, символи, низове, структури, масиви) и някои типове данни свър­зани с ООП (например класове и интерфейси).

## 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)

Всички тези типове повече или по-малко вече са ни познати от езика C#, но всъщност те са част от CTS. Езикът C# и другите .NET езици използват CTS типовете и им съпоставят запазени думи съгласно своя синтаксис. Например типът **System.Int32** от CTS съответства на типа **int** в C#, а типът **System.String** – на типа **string**.

# 42.Стойностни типове. Стандартни и user-defined.

## 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.

# 43. Елементи на типа: методи, събития, полета, properties. Примери.

# 44.Събития. Кратък пример.

## Събития

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.

## Изпращачи и получатели

Обектът, който предизвиква дадено събитие се нарича **изпращач на събитието (event sender)**. Обектът, който получава дадено събитие се нарича **получател на събитието (event receiver)**. За да могат да получават дадено събитие, получателите му трябва преди това да се абонират за него (subscribe for event).

За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите на събитието, което той предизвиква. Затова чрез механизма на събитията се постига по-ниска степен на свързаност (coupling) между отделните компоненти на програмата.

## Събитията в .NET Framework

В компонентния модел на .NET Framework абонирането, изпращането и получаването на събития се поддържа чрез делегати и събития. Реализа-цията на механизма на събитията е едно от главните приложения на делегатите. Класът, който публикува събитието, дефинира делегат, който абонатите на събитието трябва да имплементират. Когато събитието бъде предизвикано, методите на абонатите се извикват посредством делегата. Тези методи обикновено се наричат **обработчици** на събитието. Делега-тът е multicast делегат, за да могат чрез него да се извикват много обра-ботващи методи (на всички абонати).

И звикването на събитие може да стане само в класа, в който то е дефинирано. Това означава, че само класът, в който се дефинира събитие, може да предизвика това събитие. Това е наложително, за да се спази шаблонът на Публикуващ/Абонати – абонираните класове се информират при промяна на състоянието на публикуващия и именно публикуващият е отговорен за разпращане на съобщенията за промяната, настъпила у него.

## 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();

}

}

# 45.Проектиране на тип, предлагащ събитие. Проектиране на тип, използващ събитие. Същността на нещата.

Tип с дефинирани в него събития уведомява други обекти за възникването им.  
Други обекти могат да регистрират или отрегистрират интерес към събитието, като го обработят по различен начин.

Проектиране на тип, предлагащ събитие

class EventManager {

*// 1.следва вграден тип, дефиниращ информацията, предавана на*

*// получателите на събитие*

public class MailMsgEventArgs : EventArgs {

public MailMsgEventArgs( String from, String to, String subject, String body)

{ this.from = from; this.to = to; this.subject = subject; this.body = body;}

public readonly String from, to, subject, body; }

// *2.следва делегат, дефиниращ прототип на callback метод, който*

*// получателите следва да имплементират*

public delegate void MailMsgEventHandler ( Object sender, MailMsgEventArgs args);  
 *//3. Следва дефиниция на самото събитие (получателите да импл. такъв callback метод)*public event MailMsgEventHandler MailMsg;  
 *// 4. метод, отговорен за уведомяване на регистриралите интерес към събитието обекти*  
protected virtual void OnMailMsg(MailMsgEventArgs e) *//може да се предефинира поведението*{ if(MailMsg != null) *//има ли регистрирали интерес към събитието*{MailMsg( this. e); *//уведомяваме всички рег. обекти* }  
 }  
*// 5. метод, получаващ от вход данни и ги превежда. Възбужда събитието*public void SimulateArrivingMsg(String from, String to, String subject, String body)  
 { MailMsgEventArgs e = new MailMsgEventArgs(from, to, subject, body);  
 //вика метода уведомяващ обектите за събитието  
 OnMailMsg(e);}}

Б. проектиране на тип, слушащ за събитие

class Object1

{

// подаване като параметър в конструктора на обекта със събитието (EventManager в случая )

public Object1(EventManager mm)

{ // добавяме референция към списъка слушатели на събитието MailMsg (сега това е callback метода

// с име Object1Msg и имащ същата сигнатура като създадения в класа EventManager

// делегатен тип – MaiMsgEventHandler )

mm.MailMsg += new EventManager.MailMsgEventHandler(Object1Msg);

// конструира се делегатен обект, обвиващ сега метода Object1Msg като вътрешно се вика

// mm.add\_EventManager(new EventManager.MailMsgEventHandler(Object1Msg)) за регистрация

}

// следва описание на callback метода , който EventManager ще извика при събитието

private void Object1Msg( Object sender, EventManager.MailMsgEventArgs e)

{…………………………….}

public void Unregister( EventManager mm)

{

// конструираме инстанция на MailMsgEventHandler делегата, рефериращ callback метода

// Object1Msg и го отрегистрирваме като елемент от списъка.

// С# не допуска директно викане на add и remove, но от езици без събития – е възможно

EventManager.MailMsgEventHandler callback =

new EventManager.MailMsgEventHandler(Object1Msg);

mm.MailMsg -= callback;

//вика вътрешно mm.remove\_MailMsg(callback)

}

}

# 46.Пакетирани типове (boxed types). 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.

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.

## Особености при опаковането и разопаковането

При използване на автоматично опаковане и разопаковане на стойности трябва да се имат предвид някои особености:

- Опаковането и разопаковането намаляват производителността. За оптимална производителност трябва да се намали броят на опакованите и разопакованите обекти.

- Опакованите типове са копия на оригиналните стойности, поради което, ако променяме оригиналния неопакован тип, опакованото копие не се променя.

При работа с опаковани обекти трябва да се внимава, защото ако не бъдат съобразени някои особености, може да се наблюдава странно поведение на програмата . Основната причина за този резултат е фактът, че при преобразуване към интерфейс структурите се опаковат и съответно се създава копие на данните, намиращи се в тях. Опаковането е съвсем в реда на нещата, като се има предвид, че структурите са стойностни типове, а интерфейсите са референтни типове. Всеки box type поддържа интерфейси

using System;

struct Point

{

public Int32 x,y;

public void Change(Int32 x, Int32 y)

{ this.x = x; this.y = y; }

public override String ToString()

{ return Strinf.Format( “{0}, {1}”, x, y);}

}

class App

{

static void Main()

{

Point p = new Point();

p.x = p.y = 1;

Console.WriteLine(p);// (1,1)

p.Change(2, 2);

Console.WriteLine(p);// (2, 2)

Object o = p;// boxing

Console.WriteLine( o );(2, 2)

(( Point) o).Change(3, 3);// защото Change() е метод на Point

// “ о “ се разопакова. Полетата му се копират във value type Point в стека и се променят на 3

Console.WriteLine( o );// (2, 2) – това е “пакетираният” обект

# 47.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.

Reference Types

\* комбинира инф. за местоположение и за съдържание.

\* местоположението е “Type Safe”, т.е. само assignment-compatible типове могат да се пазят там.

\* garbage collector мести тези обекти свободно.

А. Object Types

наследяват Object (System.Object).

Основни методи: Equals; Finalize (вика се преди gc да освободи паметта); GetType (важно е че методът е невиртуален – т.е. не е възможно някой да се представи за друг); MemberwiseClone; ToString.

Важен CLR вграден тип (освен Object) е String

\* той е sealed

\* всеки метод, модифициращ низ, създава нов низ

Референтни типове (типове по референция) са указателите, класовете, интерфейсите, делегатите, масивите и опакованите стойностни типове. Физически референтните типове представляват указател към стойност в динамичната памет, но за CLR те не са обикновени указатели, а специални типово обезопасени указатели. Това означава, че CLR не допуска на един референтен тип да се присвои стойност от друг референтен тип, който не е съвместим с него (т.е. не е същия тип или негов наследник). В резултат на това в .NET езиците грешките от неправилна работа с типове са силно намалени.

## Референтните типове и паметта

Всички референтни типове се съхраняват в **динамичната памет** (т. нар. **managed heap**), която се контролира от системата за почистване на паметта (garbage collector). Динамичната памет е специално място от паметта, заделено от CLR за съхранение на данни, които се създават динамично по време на изпълнението на програмата. Такива данни са инстанциите на всички референтни типове.

Когато инстанция на референтен тип престане да бъде необходима на програмата, тя се унищожава от системата за почистване на паметта (т. нар. garbage collector).

Когато инстанцираме референтен тип с оператора **new**, CLR заделя място в динамичната памет, където ще стоят данните и един указател в стека, който съдържа адреса на заделеното място. Веднага след това заделената памет се занулява (освен ако програмистът не инициализира заделената променлива, например чрез извикване на подходящ конструктор).

Ако референтен тип (например клас) съдържа член-данни от стойностен тип, те се съхраняват в динамичната памет. Ако референтен тип съдържа член-данни от референтен тип, в динамичната памет се заделят указатели (референции) за тях, а техните стойности (ако не са **null**) също се заде-лят също в динамичната памет, но като отделни обекти.

## Референтните типове и производителността

Ако изчислим средното време, необходимо за заделяне и освобождаване на динамична памет, се оказва, че заделянето и освобождаване на стой-ностните типове е значително по-бързо от референтните типове. Когато производителността е важна за нашата система, трябва да се съобразя-ваме с особеностите на стойностните и референтните типове и начина, по който те заделят и освобождават памет.

Глобално погледнато, нещата около управлението на динамичната памет в .NET Framework са доста комплексни, но в тази тема няма да се спираме на тях. По-нататък, в темата за управление на паметта и ресурсите, ще им обърнем специално внимание.

# 48.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

Interface Types

Base Framework ( а не CLR) поддържа множество интерфейсни типове

using System;

using System.Collection;

namespace StringSrray

{

class EntryPoint

{

static void Main(string[] args)

{String [] names = {“ Nakov”, Angelov”, Najdenov”};

// автоматично се създава array type , имплементиращ IEnumerator

// типът поддържа редица методи: Clear(); GetLenght(); Sort()IsSynchronized(), SyncRoot()…

// Sort() използва IComparable, който следва да е реализиран от array типа ни

IEnumerator I = names.GetEnumerator();

while(i.MoveNext())

Console.WriteLine(i.Current);

}}}

Интерфейси и Value типове

using System;

namespace InterfaceSample

{

public delegate void Changed();

interface IPoint

{

int X { get; set;}

int Y { get; set;}

}

struct Point : IPoint // value type , наследил интерфейс

{ private int xValue, yValue;

public int X

{ get { return xValue;}

set { xValue = value;}

}

public int Y

{ get { return yValue;}

set { yValue = value;}

}

}

public class EntryPoint

{ public static int Main()

{

Point p = new Point();

p.X = p.Y = 33;

IPoint ip = p; // interface pointer – в heap, value типа в стека

Console.WriteLine(“X: {0}, Y: {1}”, ip.X, ip.Y);

// ip има достъп само до reference типове, докато p е value type. Следователно p е опакован и

// ip има достъп до опакования обект и членове.

}

}}

Интерфейсите описват функционалност (група методи, свойства, индекса­тори и съби­тия), която се поддър­жа от множество обек­ти. Подобно на класовете и структурите те се състоят от членове, но се различават от тях по това, че дефинират само прототипите на чле­но­ве­те си, без конкретната им реали­зацията.

От интерфейсите не могат да се създават обекти чрез инстанциране. Интерфейсите се **реализират** от класове или структури, които имплемен­тират всички дефинирани в тях членове. Конкретните имплементации на даден интерфейс вече могат да се инстанцират и да се присвояват на про­мен­ливи от тип интерфейс. Интерфейсите могат да съдържат методи, свойства, индексатори и събития. В интерфейс не могат да се дефинират конструктори, деструк­то­ри, полета и вложени типове и не могат да се предефинират оператори. Интерфейсите в C# не могат и да съдържат и константи, за разлика от дру­ги обектно-ориентирани езици, като Java, където това е допустимо. Към членовете на интерфейс не може да се прилагат модификатори на достъпа – по подразбиране всички членове са с глобална видимост, все едно е указан мо­ди­фикатор **public**. Интерфейс може да наследи един или повече други интерфейса, като е възможно да предефинира или скрива техните членове. За пример да разгледаме няколко дефиниции на интер­фейси:

interface IMovable

{ **void** Move(int aDeltaX, int aDeltaY); }

**interface** IShape

{ **void** SetPosition(int aX, int aY);

**double** CalculateSurface(); }

**interface** IPerimeterShape : IShape

{ **double** CalculatePerimeter(); }

**interface** IResizable

{ **void** Resize(int aWeight);

**void** Resize(int aWeightX, int aWeightY);

**void** ResizeByX(int aWeightX);

**void** ResizeByY(int aWeightY); }

**interface** IDrawableShape : IShape, IResizable, IMovable

{

**void** Delete();

**Color** Color { **get**; **set**; }

}

Дефинирахме следните интерфейси: **IMovable**, **IShape**, **IPerimeterShape**, **IResizable** и **IDrawableShape**. Те илюстрират дефинирането на методи и свойства в интерфейс, както и наследяването между интерфейси (което може да бъде и множествено, както е например при **IDrawableShape**).

## Интерфейсът IComparable

Често пъти освен за равенство е необходимо обектите да се сравняват спрямо някаква подредба (например лексикографска за низове или по големина за числови типове). В .NET Framework типовете, които могат да бъдат сравнявани един с друг, трябва да имплементират интерфейса **System.IComparable**.

Интерфейсът дефинира един-единствен метод – **CompareTo(object)**. Този метод трябва да реализира сравняването и да връща:

- **число < 0** – ако подаденият обект е по-голям от **this** инстанцията

- **0** – ако подаденият обект е равен на **this** инстанцията

- **число > 0** – ако подаденият обект е по-малък от **this** инстанцията

**IComparable** се използва от .NET Framework при сортиране на масиви и колекции и при някои други операции, изискващи сравнение по големина.

## Системни имплементации на IComparable

**IComparable** е имплементиран от много системни .NET типове, като например от примитивните стойностни типове **System.Char**, **System.Int32**, **System.Single**, **System.Double**, от символните низове (**System.String**) и от изброените типове (**System.Enum**). Това улеснява разработчиците при всекидневната им работа и често пъти им спестява излишни усилия.

## Интерфейсите IEnumerable и IEnumerator

В програмирането се срещат типове, които съдържат много на брой ин-станции на други типове. Такива типове се наричат **контейнери** или още **колекции**. Колекции например са масивите, защото съдържат много на брой еднакви елементи.

Често пъти се налага да се обходят всички елементи на даденa колекция. За да става това по стандартен начин, в .NET Framework са дефинирани интерфейсите **IEnumerable** и **IEnumerator**.

**Интерфейсът IEnumerable**

Интерфейсът **System.IEnumerable** се имплементира от колекции и други типове, които поддържат операцията "обхождане на елементите им в някакъв ред". Този интерфейс дефинира само един метод – методът **GetEnumerator()**. Той връща итератор (инстанция на **IEnumerator**) за обхождане на елементите на дадения обект.

Обектите, поддържащи **IEnumerable** интерфейса, могат да се използват от конструкцията **foreach** в C# за обхождане на всичките им елементи.

Интерфейсът **IEnumerable** е реализиран от много системни .NET типове, като **System.Array**, **System.String**, **ArrayList**, **Hashtable**, **Stack**, **Queue**, **SortedList** и др. с цел да се улесни работата с тях.

**Интерфейсът IEnumerator**

Интерфейсът **System.IEnumerator** имплементира обхождане на всички елементи на колекции и други типове. Той реализира прост итератор чрез следните методи и свойства:

- Свойство **Current** – връща текущия елемент.

- Метод **bool MoveNext()** – преминава към следващия елемент и връща **true**, ако той е валиден.

- Метод **Reset()** – премества итератора непосредствено преди първия елемент (установява го в начално състояние).

# 49. 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:

# 50.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.

Чрез тях се осъществяват "обратни извиквания" (callbacks). Могат да сочат както към статични методи, така и към методи на инстанция. Делегатите в .NET Framework са специални класове и могат да бъдат два вида:

Съществува известна прилика между делегатите и указателите към функ-ции в други езици, например Pascal, C, C++, тъй като последните пред-ставляват типизиран указател към функция. Делегатите също съдържат силно типизиран указател към функция, но те са и нещо повече – те са напълно обектно-ориентирани. На практика делегатите представляват класове. Инстанцията на един делегат може да съдържа в себе си както инстанция на обект, така и метод.

Едно от основните приложения на делегатите е реализацията на "обратни извиквания", т.нар. callbacks. Идеята е да се предаде референция към метод, който да бъде извикан по-късно. Така може да се осъществи например асинхронна обработка – от даден код извикваме метод, като му подаваме callback метод и продължаваме работа, а извиканият метод извиква callback метода когато е необходимо. Със средствата на делега-тите е възможно даден клас да позволи на потребителите си да предоставят метод, извършващ специфична обработка, като по този начин обработката не се фиксира предварително.Делегатите в .NET Framework са специални класове, които наследяват **System.Delegate** или **System.MulticastDelegate**. От тези класове обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за да се наследяват от тях типове делегат.

Всеки делегат има **"списък на извикване" (invocation list)**, който представлява наредено множество делегати, като всеки елемент от него съдържа конкретен метод, рефериран от делегата.

# 51. 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");

}

}

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

# 52. 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.