**Езикови средства за създаване на устойчив код. Въведение в SEH.**

SEH(Structure Exception Handling) структурирана обработка на изключителни ситуации. Това е вграден механизъм за обработка на софтуерни(прекъсвания) и хардуерни(деление на нула) изключителни ситуации.Пример:  
***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: използващ предоставените от средата средства за справяне с изключителни ситуации, няма толкова проверки, добре блокуван и се знае кое кога ще се изпълни.  
***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 втория случай.   
 Блокът след exception handler ще се изпълни при наличие на exception в try. Блокът exception се изпълнява в обсега на try.  
 В 32 битова среда има следните възможности за прихващане и обработка на exception:  
1.Може да се ползва SEH механизма на ОС.  
2.Може да се ползва собствения за езика механизъм за обработка на exceptions.  
3.Възможна е комбинация то 1 и 2.  
При откриване на exception става:  
1.Изпълняваната нишка се прекратява.  
2.Управлението се предава от usermode в kernelmode (ядрото на ОС поема управлението).  
3.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.  
1.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;}  
 return fReturn;  
}***  
 finally предшества блок, който винаги ще се изпълни и то преди края на функцията. Ако имаме и finally и exception, то finally се изпълнява след exception блока. Finally се изпълнява в следните случаи:  
1) след изход от try секция;  
2) след прекратяването на try блока чрез exception;  
3) при нормален изход;  
 Return по възможност да е извън try и finally блоковете. Иначе return може да е в try, finally или извън 2та блока. Ако е в try блока връщаната стойност не се връща в този момент, т.е компилатора съхранява резултата от връщане в междинна памет и ще го върне след като handler-а се изпълни. Това означава, че ако след finally блока има още 1 return той няма да върне стойност.Пример:  
***Long calcFactorial(int n)  
{   
 long lResult;  
 \_\_try   
 {  
 if (n<2) { for(lResult=1;n>1;\_\_)lResult\*=n;}  
 else return(long)n;  
 }  
 \_\_finally{ return lResult;}  
}*** Винаги се изпънява Return от finally, което е грешно.  
**\_\_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. Единственият начин да не се изпълни finally е да прекратим нишката(abroad прекъсване).

**Филтър за обработка на изключителни състояния (\_\_except).**

Той се изчислява и стойността определя начина на реакция при случване на exception.  
**1)EXCEPTION\_EXECUTE\_HANDLER – term.handler** – има стойност 1:Ще се изпълни блокът след except и управлението ще се подаде след блока except.  
**2)EXCEPTION\_CONTINUE\_SEARCH (0)** –dispatcher ще търси друг. Има стойност 0. Той показва на ОС да пренебрегне нашият exception блок и да търси обхващащ exception блок.  
**3)EXCEPTION\_CONTINUE\_EXECUTION** –пренебрегва (-1) - има стойност –1: Той пренебрегва exception блока т.е. той не се изпълнява, няма да се търси друг, а управлението ще се предаде на следващата конструкция.

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;   
}***

**Вграден в С++ механизъм за р-я на изключителни събития**

C++Exception Handling  
***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 няма catch, то се търси в обхващащата функция дали има catch, ако никъде няма то се извиква terminate. При генериране на изключителни ситуации с throw автоматично се изпълняват деструкторите на всички обекти създадени от началото на try блока.  
**Standart Exception Library**  
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, к. не могат да се групират при изпълнението  
**- logic error** - групира всички проблеми, к. се пораждат при изпълнението 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 чрез try. Конструкторът, не връща стойност т.е. не може да информира за успех или неуспех относно работата му.  
***{cout << “ ……sssdcg “ << endl;  
throw; }}***

Windows отделя нишка, която се занимава с анимации.

**Изключения и .NET**

Изключението не е грешка. То е просто нарушаване на допусканията за даден програмен интерфейс.  
Отказ от **HRESULT:**-Изключение не може да се игнорира  
-няколко десетки стандартни типа на изключения, всички наследили System.Exception  
-стек на повикванията за catch  
-ако липсва филтър – генерира се “необработено съобщение”  
След откриване на подходящ catch, изпълняване на всички finally от блока, подал изключението, докато стигне ( и там спира ) блока с catch филтъра, поел го за обработка.. След това се изпълнява catch блока.  
В края на catch блока:   
а) подаване същото изключение нагоре;   
б) подава се друго изключение с по-богата информация нагоре;   
в) нишката излиза от catch.  
**“Трагични” съобщения:** OutOfMemoryException; StackOverflowException; ExecutionEngineException  
**System.Exception** съдържа полезни свойства: **Message; Source; StackTrace; TargetSite; HRESULT**.

**Обща йерархия в .NET Framework Class Library (FCL):**

System.Exception= System.ApplicationException==System.SystemException

Подавайте конкретен клас изключение ( който няма производни класове). Йерархиите exceptions да са широки , но плитки. Тези без производни – sealed.  
 **Дефиниране на собствено изключение:**

1.да има 3 public конструктора:  
**а.** без параметри (по подразбиране)  
**б.** с параметър string – задаващ съобщение  
**в.** с параметър String и инстанция на производен на Exception тип ( за генерирано вътрешно съобщение)  
2. наследява Exception  
3. може да има и други констуктори  
4. винаги да е сериализуем за да може изключението да се предава през граница на процес, машина или кл. приложение или в log. За целта:  
- [Serializable] атрибут  
- ако има дефинирани свои даннови членове: - да е наследил ISerializable  
- да имплементира метода му GetObjectData()  
- да има констр. с 2 парам. – SerializationInfo и StreamingContext  
[Serializable]  
***sealed class MyPrivateException : Exception, ISerializable {  
 public MyPrivateException() : base() {}  
 public MyPrivateException(String message): base(message) {}  
 public MyPrivateException(String message, Exception innerException);   
 base (message,innerException) { }// дефинира собствено поле private String name;  
// дефиниране на свойство само за четене, което връща данната public String Name Ш get{return name; } } //предефиниране базовото свойство Message, така че да включва и новото поле public override String Message  
{   
 get { String msag = base.Message;  
 msg + = “ The name is” + name;  
 return msag; }}// тъй като имаме поле, дефинираме конструктор за десериализация.***

***// Той е private, защото класа е sealed, иначе – protected private MyPrivateException (SerializationInfo info, StreamingContext context): base( info, context)  
{ /\* десериализация на полетата на базовия конструктор \*/  
// десериализация на всяко ново поле  
name = info.GetString(“Name”);}  
// тъй като има поне 1 метод или данна, предефинираме метода за сериализация  
void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context) {  
// сериализация на всяко поле  
info.AddValue(“Name”, name);  
base.GetObjectData(info, context);}  
// дефиниране допълнителни конструктори, задаващи стойност на полето   
public MyPrivateException(String message, String name): this (message)  
{  
 this.name = name;  
}  
public MyPrivateException( String message, String name, Exception innerException): this(message, innerException)  
{ this.name = name;}***

**Правила за работа с изключения**

Разработвате библиотека: ако прихванете всички изкл. , как разработващия приложение с библиотеката ще знае че нещо се е случило. Разработвате библиотека с типове – не винаги знаете кое е грешка, кое не. Избягвайте код, прихващащ всичко: ***catch(System.Exception) {………}*** 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 потребителя).  
 Внимание: CLR управлява изключенията така: първо намира catch филтър, който може да е доста нагоре в стека. След това “развива” стека на вложените изкл. като изпълнява всички Finally блокове , до и без този съответстващ на прихващащия филтър.Следва изпълнение на catch блока и след това неговия finally.