Вход/изход и сериализация с поддръжка на MFC среда

Работа без сериализация – базов клас CFile

(над 25 метода, 1 public променлива – m_hFile; 1 protected данна – CSting m_strFileName и т.н.)

```
CFile myfile;
CFileException e;
if( file.Open( _T("My.txt"), CFile::modeReadWrite, &e));
  {
                // работим с файла
  }
else
  {   ……
      e.ReportError();
  }
```

същото може и така:

```
try {
       CFile file ( _T("MyFile.txt"), CFile::modeReadWrite);
                  …..
       }
catch( CFileException& e)
    {
      e→ReportError();
      e→Delete();
     }
```

затваряне:

file.Close();

CFile обект се затваря автоматично 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; } // оставащите байтове

- file.Write(buff, nByteRead); // записва определен брой байтове от буфер във файл
- file.Seek(относително_отместване в байтове, спрямо ..)

препоръчително е четенето да се обхване от try…catch

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

Производни на CFile класове (осигуряват файл-подобен интерфейс за програмиста)

CMemFile, CSharedFile // работи с блокове памет, аналогично на файл

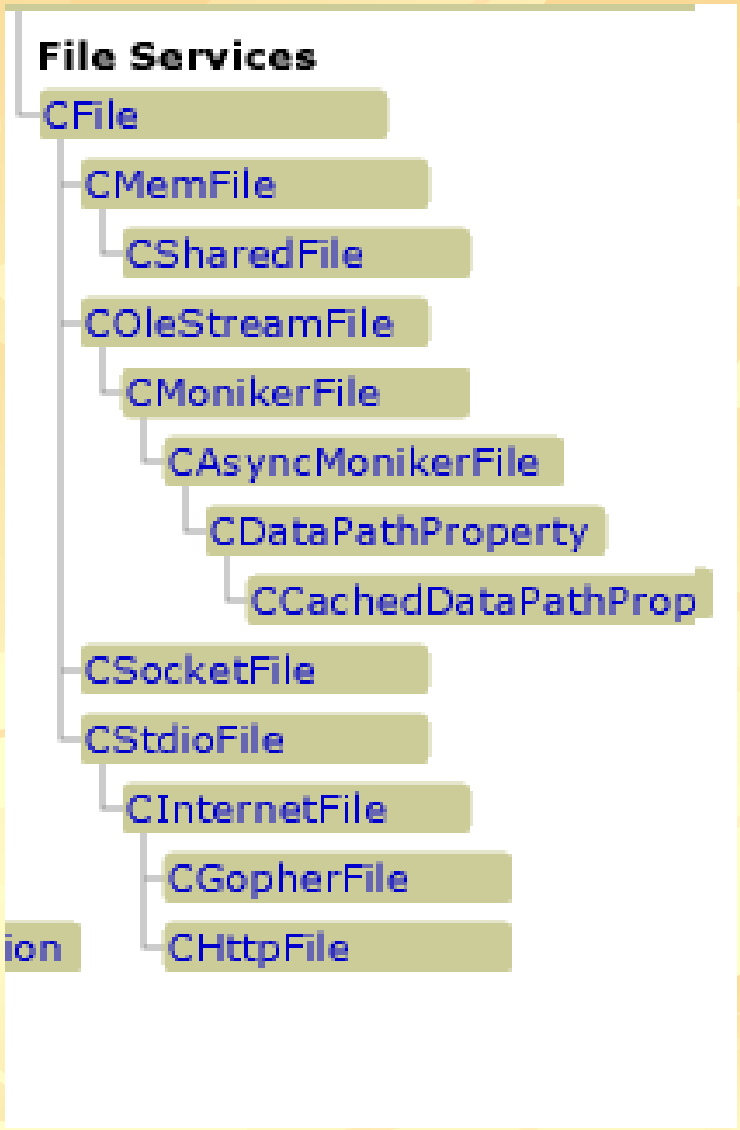
COleDataObject::GetFileData() // асоциира OLE clipboard с файл

CSocketFile //стои между CArchive и CSocket обекта. Тогава << и >> работят към отворен socket

CStdioFile // наследник на CFile, с интерфейс за текстови файлове. Пример:

```
try {
    CStdioFile file( _T("My.txt"), CFile::modeRead);        …..}
/* (възможни са комбинации от режими:CFile::modeCreate, CFile::shareDenyRead, CFile::shareDenyWrite,
                                        CFile::shareDenyNone .. ) */
catch( CFileException* e) {        ..        }
```

CInternetFile, CGopherFile и CHttpFile .


File Services
CFile
CMemFile
CSharedFile
COleStreamFile
CMonikerFile
CAsyncMonikerFile
CDataPathProperty
CCachedDataPathProp
CSocketFile
CStdioFile
CInternetFile
CGopherFile
CHttpFile

Изброяване на файлове и директории

::FindFirstFile - позиционира на първия файл в директорията. Изработва file_handle

:: FindNextFile() - попълва стандартна структура с атрибути, име, размер и др. за файла включително и дали е файл или директория.

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

(всеки обект е отговорен за съхраняване/възстановяване на своето състояние. Архивът стои между обекта и запомнящата за него среда. Така се абстрахираме от особеностите по съхраняване и работим по унифициран начин)

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

```
file.Write( &a, sizeof(a));        file.Write( &b, sizeof (&b));
```

Ето другия подход:

```
CArchive ar( &file, CArchive::store);    ar << a << b;
```

всички примитивни типове имат предефинирани в MFC операции за '<<' '>>'
(BYTE, WORD, LONG, DWORD, float, double, int, short, char, unsigned int).
Операторите са предефинирани и за непримитивни типове, за които има стандартни MFC класове.
Пример:

```
CString string;        ar << string;
```

също и за: CTime, CRect, CSize, CPoint, ColeDateTime, COleVariant (OLE Automation типове)…,
както и структури SIZE, POINT, RECT…
(операция - сериализиране е предифинирана за стойности,
десериализация →за указатели)

Голямото предимство на сериализацията в MFC среда е че можете ДА СЪЗДАВАТЕ СВОИ, ПРИ ТОВА СЕРИАЛИЗИРАНИ КЛАСОВЕ, БЕЗ ДА ПРЕДЕФИНИРАТЕ << И >>.

MFC автоматично добавя код за това, стига да става дума за сериализация на указатели към класове, производни на CObject.

Пример:
имате клас линия. Имате масив линии и искате да съхраните елементите му. Щом е наследил CObject, имате право да използвате директно <<

Ако се случи грешка – MFC изработва обект CArchiveException

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, ползван при динамичното му създаване при >>. Това е възможно, благодарение на информацията от**
- **В частта имплементация на класа IMPLEMENT_SERIAL**

```
class CLine : public CObject
{                   DECLARE_SERIAL (CLine)
protected:                  CPoint m_ptFrom;                    CPoint m_ptTo;
public:                     CLine()     {}          //задължителен, дори и празен
      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);  //сериализираме данните на базовите класове. За CObject
                                              //  това може да се избегне
      if ( ar.IsStoring())
                    ar << m_ptFrom << m_ptTo;
      else
                    ar >> m_ptFrom >> m_ptTo;                    }
```

Добавяте и някъде в имплементацията:

IMPLEMENT_SERIAL(CLine, CObject, 1) //версиите следва да са еднакви.
//Иначе – CArchiveException. Ar.GetObjectSchema()

(IMPLEMENT_SERIAL(CLine, CObject, 1 | VERSIONABLE_SCHEMA))

Интересна е и още една особеност:

Предефинирането на <<, >> работи за указатели към такива обекти-наследници, не с обекти :

```
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
```

ако 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; // ако  операция "=" е предефинирана в CLine като копиране
delete pLine;           // вече ненужен
```

Същото може да се постигне (ако държите да работите с обекти, а не указатели) с директно викане на Serialize() (за вложен обект – напр. CPoint):

```
CLine line(CPoint(0,0), CPoint(40,40));
line .Serialize(ar);                    //за сериализация

CLine line;
line.Serialize(ar);      // за десериализация
```

Сега, обаче липсва информация за версия

Как работи сериализациацията

- Класът CArchive още веднаж предефинира операцията <<. Винаги се конструира междинен вътрешен буфер в който се записват сериализираните данни. При запълването му – те се прехвърлят към свързания с архив обект (CFile напр).
- При повечето типове (напр. CString) в буфера се записва и доп. Информация: напр. дължина, кодировката. (това позволява сериализиране в ANSI и десериализиране в UNICODE и обратно.

- **За сериализиране на обекти** се вика **ar.WriteObject(pObj)**, който метод от своя страна вика **Serialize()** на обекта. Преди да се сериализират данните на обекта обаче, се съхранява и допълнителна информация:
  - Дали обектът е първичен или има и други съхранени от този клас;
  - Номер на проект;
  - Име на базовия класови обект.

  Втори записан обект от същия тип, получава служебен индекс и не съхранява служебна информация.

  Той е old class

*** За десериализиране на обекти:**

- операторът >> има информация колко обекта от конкретния клас са съхранени и може да итерира през тях;
- самата десериализация се изпълнява от вътрешно повикване на **ar.ReadObject()**, викаща последователно:
  - -- **ReadClass()** – за определяне типа на обекта (от съхранените данни в структура **CRuntimeClass**, създадена и попълнена авоамтично) и номера на проекта
  - -- **CreateObject()** за създаване нов екземпляр от класа;
  - -- **Serialize()** за зареждане данновите му членове от архива.

Вградени обекти в клас за който обекти сме реализирали сериализация

имаме: class CStudent { …….. public: CDisc m_disciplines;….}

Вграден клас със своя сериализация, производен на CObject

1. CStudent::Serialize(CArchive& ar)
{if(ar.isStoring()) { ar << data1<< data2;}
else { ar >> data1 >> data2; }
m_disciplines.Serialize(ar); }

Инстанция от CDisc вече е създадена (в констр на CStudent).

Викаме Serialize() за всеки вграден обект, проиводен на CObject

не е допустимо: ar >> data1 >> data2 >> &m_disciplines;
//защото оп << и >> не са предефинирани за стойност, а за указател, докато нашата инстанция е по стойност

2. имаме: class CStudent { …….. public: CDisc* m_pdisciplines;….}

Имаме даннов член – указател към наш клас

тогава: CStudent::Serialize(CArchive& ar)
{ if(ar.isStoring()) { ar << data1<< data2;}
else { m_pdisciplines = new CDisc; ar >> data1 >> data2; } // инициализираме указателя
m_pdisciplines→Serialize(ar); } // викаме Serialize()

или:

3.

```
CStudent::Serialize(CArchive& ar)
{
   if(ar.isStoring())
            ar << data1<< data2 << m_pdisciplines;      //  имаме указател, към вече създаден обект
           else                                          // и директно сериализираме.
            ar >> data1 >> data2 >> m_pdisciplines; }
```

// как се е създал обекта при десериализацията? При сериализирането се е записала информация и за
//класа. Макросите DECLARE_SERIAL, IMPLEMENT_SERIAL вършат останалото по създаването.
//Класът беше призводен на CObject, така че той работи по подразбиране със сериализация

Runtime Serialization (.NET)

Пролемът е : преобразуване на обект или групирани обекти в stream of bytes и обратно.
.NET решава въпроса за почти всички типове (почти) автоматично.

```
using ….
 using System.Runtime.Serialization.Formatters.Binary;

internal static class QuickStart {
 public static void Main() {

 var objectGraph = new List<String> {“Jeff”,”ABC”, “Ivan”};
 stream stream = SerializeToMemory(objectGraph);
…
objectGraph = List<Strimg>) DeserializeFromMemory(stream);
…              }
private static MemoryStream SerializeToMemory( Object objectGraph) {
// конструира обектът, който ще поеме сериализирания обект
 MemoryStream stream = new MemoryStream();
// конструира  serialization formatter обект, който поема цялата работа по преобразуването
BinaryFormatter formatter = new BinaryFormatter();

// казва на formatter да сериализира обекта в stream
 formatter.Serialize(stream, objectGraph);  …. }

 private static Object DeserializeFromMemory(Stream stream) {
  BinaryFormatter formatter = new BinaryFormatter();
 return  formatter.Deserialize( stream);
}}
```

Formatter е обект имплементирал IFormatter интерфейс. Той знае как се Сериализира/десериализира обект-граф.

CLR има 2 formatter обекта: BinaryFormatter и SoapFormatter

Референциата към обект за сериализация може да е всичко: Int32, String, Data, Exception, List<String>, и т.н.
Formatter знае как – на основа на метаданните

-В 1 formatter обекта могат да се сер/десер повече от 1 обекти.

Runtime Serialization (.NET)
- Making a type serializable

-По подразбиране обектите не са сериализуеми. Следователно повикване на обект.Serialize(stream, инстанция); ще генерира SerializationException;
За да стане дефиниция на обект сериализуема - [Serializable] атрибут;
-[Serializable] може да се добавя към : class, struct, enum, delegate
-Всички полета на сериализуем тип автоматично стават сериализуеми, независимо от модификатора за достъп.

-Понякога се налага някои полета да се изключат от сериализацията: напр mutex , който ще се десериализира в различен процес. Тогава:

```
[Serializable]
class Circle {
        private Double radius;

        [NonSerialized]
        private Double m_area;
        ….
        }
```

Как действа сериализацията:

- **Formatter извършва сериализацията. Той ползва типа FormatterService FormatterService, който има само static методи (не може инстанция) и подпомага операциите.**

1. **Вика се негов метод, който определя полетата , които подлежат на сериализация/десериализацияя;**
2. **Вика се друг метод на FormatterService който чете тези полета и попъква с тях масив**
3. **Formatter попълва в служебна информация в асемблито за типа и stream. При десериализация, тази информация се чете предхождащо**
4. **Formatter последователно записва елементите в stream**

- **Ако искаме да имаме пълен контрол как ще действа сериализацията, то следва да имплементираме**
- **System.Runtime.Serialization.ISerializable интерфейса:**

```
public interface ISerializable    {
        Void GetObjectData(SerializationInfo info, StreamingContext context
                                    }
```

както и IDeserializationCallback имащ метода: OnDeserialization(Object sender);

Този обект съдържа стойностите, които ше се серилизират. Използвайки го formatter обекта, през GetObjectData() го попълва с данните. В процеса помага и SerializationInfo.AddValue(),

Съдържа информация дали сериализацията ще се прави в друг процес, машина, файл, друго приложение и т.н.)