Когда в C# не хватает C++ vol 3

Сергей Балтийский
JetBrains

Why?

- Скорость
  - Оптимизация CPU
  - Затраты на переключение контекста
- Память
  - Управление памятью
  - GC
- Legacy
  - Библиотеки на C/C++

How?

- C++/CLI
- COM
- PInvoke
- Native Memory in C#

Storing C# Data in Native Memory

- Unmanaged Heap
  - Marshal::AllocHGlobal
- A single managed array
- Disk file mapped into memory
- Shared memory
- Memory outside 32-bit process 4GB space
- **Not** a graph of linked managed objects

Отличие от обычных структур

- Структура скрыта от GC
  - Нет memory traffic
  - Нет нагрузки при обходе графа при запуске GC
- Явный/ручной memory management
- Можно работать над locality

Lifetime

- Managed объекты работоспособны пока доступны
- Native memory в какой-то момент надо явно освободить
  - Явный момент окончания использования
  - Все клиенты знают об этом
  - В структуре данных есть средства контроля

Во время загрузки

- Возможность прочитать как один BLOB
  - Bulk file read
  - Memory-mapped file
  - C# field initializer
- Нет создания managed-объектов при десериализации
  - Performance
  - Memory traffic
- Максимальная ленивость

Приличное C# API

- Работа как с обычной структурой данных
  - А не набор C-style функций

```
iter = LevelDbInterop.leveldb_create_iterator
          (DbHandle.DbPtr, DbHandle.ReadOptionsPtr);
LevelDbInterop.leveldb_iter_seek(iter, serializedKey, new IntPtr(1));
while (LevelDbInterop.leveldb_iter_valid(iter) != 0)
{
  var pkey = LevelDbInterop.leveldb_iter_key(iter, out klen);
  LevelDbInterop.leveldb_iter_next(iter);
}
```

- Не потерять выгоду на накладных расходах API
  - Не создавать GC-объектов

Safety

- Ошибки в нативном коде
- Ошибки в работе с нативной памятью
  - Чтение за пределами данных
  - Запись в «чужую» память
- Нарушение времени жизни
  - Работа с кодом или памятью после освобождения

Реализация

- 1) Native code
- 2) C#-only

Native Code Implementation

- CPP/CLI (see vol. 1)
- COM (see vol. 1)
- PInvoke
  - Выглядит проще
  - Много справочной информации
  - Доказанная портабельность
  - Больше ручной работы
  - Все конвенции надо продумывать самостоятельно
    - Calling conventions, threading, memory ownership, lifetimes

Точка входа в DLL

- `DllImportAttribute`
  - Поиск DLL только по имени файла
  - Проблема выбора 32/64/OS-specific реализации
- `LoadLibrary + GetProcAddress + GetDelegateForFunctionPointer`
  - (или `dlopen + dlsym`)
  - Загрузка DLL по явному пути
    - В том числе Side-by-Side
    - Выбор для нужной разрядности и OS
  - Гибкий выбор функций для импорта
  - `UnmanagedFunctionPointerAttribute`

Native Pointers in PInvoke

```
[StructLayout(LayoutKind.Sequential)]
public struct POINT
{ public int x; public int y; }

[StructLayout(LayoutKind.Sequential)]
public class CPoint
{ public int x; public int y; }

[DllImport("my.dll")]
public static extern int Hit(ref POINT ppt);

[DllImport("my.dll")]
public static extern int Hit(CPoint ppt);

[DllImport("my.dll")]
public static extern int Hit(POINT *ppt);
```

C#-only Implementation

- Pros:
- Код на одном языке
- Единый codebase, нет дублирования деклараций
- AnyCPU & portable
- Нет возни с native DLLs
- Нет переключения managed/native и маршаллинга

C#-only Implementation

- Cons:
- Очень ограниченные возможности C# при работе с native memory
  - Нет ООП на blittable types
  - Невозможно использовать генерический код
- Нет возможности писать высокоуровневый код
- Нет средств контроля работы с native memory из .NET

Доступ к памяти

Readonly

Read+Write

Read-Only Data Structures

- Предварительно подготовленные данные
- Нет memory allocator
- Нет опасности испортить память
- Возможные промахи по адресам
  - Могут вызвать access violation без последствий
    - Убедиться, что они ловятся
  - Могут прочитать бессмысленные данные
    - Маркеры / magic numbers где уместно

Use Case

- Hash table с большим количеством данных

```
return new Dictionary<string, string>()
{
  { "&Aacute;", "\u00C1" },
  { "&aacute;", "\u00E1" },
  { "&Abreve;", "\u0102" },
  { "&abreve;", "\u0103" },
  { "&ac;", "\u223E" },
  { "&acd;", "\u223F" },
  { "&acE;", "\u223E\u0333" },
  { "&Acirc;", "\u00C2" },
  { "&acirc;", "\u00E2" },
  { "&acute;", "\u00B4" },
  { "&Acy;", "\u0410" },
  { "&acy;", "\u0430" },
  { "&AElig;", "\u00C6" },
```

Медленное конструирование

- JIT
- CPU time
- Memory*

Альтернативная загрузка

- C# array field with initializer
  - Быстро, но память неструктурированная

```
 internal static readonly UInt32[] HashByName = new UInt32[6377]{
0x00000002u, 0x00000021u, 0x00000C73u, 0x00000000u, 0x00000000u, 0x00000000u,
0x000003F1u, 0x4D96D5CCu, 0x000000B5u, 0x6F86802Au, 0x000029B3u, 0x43609526u,
0x00002197u, 0x538C7B14u, 0x000022FCu, 0x63104C69u, 0x00002923u, 0x00000000u,
0x00002640u, 0x61C808F1u, 0x000003BDu, 0x97AA0769u, 0x000022A8u, 0x5336850Du,
0x00002A22u, 0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,
0x00000000u, 0x00000000u, 0x00000000u, 0x0B607F82u, 0x0000229Eu, 0x80C8F570u,
0x00002247u, 0x6D685FB2u, 0x00002312u, 0x09B499B2u, 0x00002AEDu, 0x00000000u,
0x00000000u, 0x53C063E9u, 0x0000045Au, 0x6D003F87u, 0x00002AB9u, 0x00000000u,
0x03382AC6u, 0x1BDA6044u, 0x00000402u, 0x4E489470u, 0x00002ADBu, 0x00000000u,
0x0000230Au, 0x4EE64817u, 0x000003BCu, 0x00000000u, 0x00000000u, 0x7D787AE8u,
0x00000000u, 0x00000000u, 0x00000000u, 0x7D904194u, 0x00002322u, 0x49ACFEFFu,
0x0000229Bu, 0x00000000u, 0x00000000u, 0x2E7EDF8Du, 0xDD3ED835u, 0x00000000u,
0x00002AA6u, 0x271CBA76u, 0x00002261u, 0x1D42C693u, 0x000000A8u, 0x7CE2111Eu,
```

Загрузка массива в field

- Код загрузки в `.ctor` / `.cctor`
  - Вызов спец-функции над спрятанным филдом

```
ldc.i4      6377
newarr      [mscorlib]System.UInt32
dup
ldtoken     field valuetype ‘$$method0x6002768-2’
call        void RuntimeHelpers::InitializeArray(Array, RuntimeFieldHandle)
stsfld      unsigned int32[] MyClass::HashByName
```

Загрузка массива в field

- Спрятанный филд

```
.field static assembly valuetype ‘__StaticArrayInitTypeSize=25508’
‘$$method0x6002768-2’ at I_00018CD0
```

```
.data cil I_00018CD0 = bytearray (
02 00 00 00 21 00 00 00 73 0C 00 00 00 00 00 00  // ....!...s.......
00 00 00 00 00 00 00 00 00 00 00 00 62 74 3C 3D  // ............bt<=
8D 29 00 00 FA D2 BC 29 02 FB 00 00 6B C8 A0 94  // .).....)....k...
0E 23 00 00 00 00 00 00 00 00 00 00 A7 F4 38 74  // .#............8t
F1 03 00 00 CC D5 96 4D B5 00 00 00 2A 80 86 6F  // .......M....*..o
B3 29 00 00 26 95 60 43 7F 2A 00 00 00 00 00 00  // .)..&.`C.*......
00 00 00 00 58 27 02 22 F6 29 00 00 00 00 00 00  // ....X'."....)....
00 00 00 00 B9 27 3A 0B 34 25 00 00 8D BB FE 50  // .....':.4%.....P
```

Загрузка массива в field

- Спец-функция `RuntimeHelpers::InitializeArray`

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
public static extern void InitializeArray(Array array,RuntimeFieldHandle
fldHandle);
```

```
FCIntrinsic("InitializeArray", COMArrayInfo::InitializeArray,
CORINFO_INTRINSIC_InitializeArray)
```

```
FCIMPL2(void, COMArrayInfo::InitializeArray, ArrayBase* pArrayRef, HANDLE
handle)
…
#if BIGENDIAN
…
#else
        memcpyNoGCRefs(dest, src, dwTotalSize);
#endif
```

Загрузка массива в field

- Одна GC-аллокация массива
- Копирование данных блоком
- Данные доступны как массив или `fixed byte*`

Hash Table in Native Memory

- Read-only, все данные известны при построении
- Выбираем размер и хороший хеш
- Открытая адресация
- Массив структур
  - Ключ
  - Значение
  - Индекс в массиве — хеш
    - по модулю
    - + probing

Hashtable Record

- Ячейка таблицы

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
private struct Cell
{
  public UInt16 dwHash;
  public UInt16 rvaName;
  public fixed UInt16 wszValue [2];
}
```

Header | Cell

```
0x00000002u, 0x00000021u, 0x00000C73u, 0x00000000u, 0x00000000u,
0x00000000u, 0x00000000u, 0x3D3C7462u, 0x0000298Du, 0x29BCD2FAu, 0x0000FB02u,
```

Hashtable Record

```
{
  public UInt16 dwHash;
  public UInt16 rvaName;
  public fixed UInt16 wszValue[2];
}
```

```
0x3D3C7462u, 0x0000298Du
```

```
{ "&lbrkslu;", "\u298D" },
```

Unicode Character 'LEFT SQUARE BRACKET WITH TICK IN TOP CORNER' (U+298D)

Извлечение примитивных данных

- Reinterpret-Cast

```
byte* pbCells;
byte* pbCell = pbCells + nCell * sizeof(Cell);
Cell* pCell = (Cell*)pbCell;

ushort dwHash1 = pCell->dwHash;

Cell cell = *pCell;

ushort dwHash2 = cell.dwHash;
```

Извлечение примитивных данных

- Pointer arithmetic

```
Cell* pCells;
Cell* pCell = pCells + nCell;

ushort dwHash1 = pCell->dwHash;

Cell cell = *pCell;

ushort dwHash2 = cell.dwHash;
```

Извлечение примитивных данных

- Array syntax

```
Cell* pCells;

ushort dwHash1 = (*(pCells + nCell)).dwHash;
```

```
ushort dwHash2 = pCells[nCell].dwHash;
```

Хранение строк

- Фиксированной длины
  - Внутри родительской структуры

```
public fixed UInt16 wszValue[2];
```

- Переменной длины
  - RVA в BLOB со строками

```
public UInt16 rvaName;
```

    - В простом случае — поток ASCIIZ строк в UTF-16LE

```
new string((char*)(pbStrings + pCell->rvaName))
```

    - Можно добавить header для валидации, длины, хеш-кода,…

Извлечение строк

- Нужен ли нам `string` object?
  - `new string()` это нагрузка на GC
  - Interning, кеширование?
  - Достаточно hash code, equals, compare?
    - Можно реализовать прямо на `char*`

Извлечение строк

- `new string(char*)`
- Правильная кодировка
- Zero-terminated
  - Или `new string(char*, int, int)`

Частные хитрости со строками

- Чтобы меньше создавать `string`
- Потребовать уникальность хеша
  - Выбрать хороший длинный хеш
  - Работать прямо с хешами, пока нам достаточно идентификации/сравнения
- Использовать metadata token вместо type full name
  - Если строчки нужны для идентификации имён типов

Поиск в hash table

```
fixed(uint* pdwTable = XmlHtmlNamedCharacterReferencesOriginalDictionary.HashByValue)
{
  var pHeader = (Header*)pdwTable;
  byte* pCells = ((byte*)pdwTable) + sizeof(Header);
  uint hash = ch;
  if(hash == 0)
    hash = 1;
  uint nBaseRecord = hash % pHeader->NumCells;
  for(uint nProbe = 0; nProbe < pHeader->NumCells; nProbe++)
  {
    uint nProbeRecord = (nBaseRecord + nProbe * ProbeFactor) % pHeader->NumCells;
    var pCell = (Cell*)(pCells + nProbeRecord * sizeof(Cell));
    if(pCell->dwHash == 0)
      return null; // Found a free cell
    if(pCell->wszValue[0] == ch)
    {
      fixed(uint* pdwStrings = XmlHtmlNamedCharacterReferencesOriginalDictionary.Strings)
        return new string((char*)(((byte*)pdwStrings) + pCell->rvaName));
    }
  }
}
return null;
```

Загрузка данных с диска

- Последовательное чтение
  - Максимально быстрое чтение
  - Занимает память
- Отображение файла в память
  - Memory Mapped Files

Memory-Mapped Files

- 1) Отображение файла на память процесса
  - Весь файл сразу доступен как native memory
  - Лениво читается с диска
  - Не увеличивает committed memory
    - Но тратит virtual address space
  - При нехватке памяти использует свой файл вместо свопа
- 2) Расширение памяти процесса за счёт переключения страниц

Memory-Mapped Files

- 2) Расширение памяти процесса за счёт переключения страниц
  - В память процесса по очереди отображаются разные небольшие участки MMF
    - Bank switching
  - Либо файл на диске, либо system swap
  - В 64-bit OS памяти много, а у 32-bit процесса мало
    - Теоретически, так можно обращаться к памяти выше 4GB
    - Но система может начать агрессивно свопить эту память

Memory-Mapped Files

- Memory-mapped files

  +

- Native memory data structure

  =

- Zero-load time data structure

Пример структуры данных

- Выжимка из метаданных
- В готовом виде для создания component container
- Быстрая фильтрация
  - По атрибутам
  - По модуль-зонам
- Удобный API
  - В терминах Assembly / Type / Member / Attribute
- Быстрая загрузка
- Без нагрузки на GC при обходе структуры

Формат данных

- POD-таблицы
  - Массив повторяющихся структур с примитивными данными
    - Строки в виде RVA, поэтому тоже примитивные данные
  - Доступ по индексу

```
Struct* _pTable;
int value = _pTable[25].IntValue;
```

Формат данных

- Нетабличные данные
  - Поток данных переменного размера
  - Произвольные объекты
  - Строки переменной длины
  - Хеш-таблицы

Table Record Example

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
public struct MemberRecord
{
  public StringRef LocalName;

  public TypeRef DeclaringType;

  public AttributeRangeRef Attributes;

  public TypeRef ValueType;

  public TypeListRangeRef ParameterTypes;

  public PartCatalogTypeMemberKind Kind;
}
```

Table Record Example

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
public struct TypeRef
{
  public UInt16 Index;
}
```

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
public struct StringRef
{
  public uint Rva;
}
```

Table Record Example

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
public struct TypeListRangeRef
{
  public ARangeRef Range;
}
```

```
[StructLayout(LayoutKind.Sequential, Pack = 2)]
public struct ARangeRef : IEquatable<ARangeRef>, IComparable<ARangeRef>
{
  public UInt16 FirstIndex;

  public UInt16 Count;
}
```

Table Example

	A	B	C	D	E	F	G
1	Index	LocalName [StringRef]	DeclaringType [TypeRef]	Attributes [Attr	ValueType [TypeRef]	ParameterTypes [T	Kind [PartCatalog
2	NULL	[NULL]	NULL	NULL	NULL	NULL	0
3	0001h	[00001E44h] .ctor	[0001h] FooImpl – JetBrains.Platfor	NULL	[001Eh] Void – System.Void	NULL	Constructor
4	0002h	[00001E44h] .ctor	[0002h] TestComponent1 – JetBrains.	NULL	[001Eh] Void – System.Void	NULL	Constructor
5	0003h	[00001E44h] .ctor	[0004h] TestComponent2Impl – JetBra	NULL	[001Eh] Void – System.Void	NULL	Constructor
6	0004h	[0000211Ch] DoJob	[0005h] TestComponentWithMembers –	0009h ∑1	[001Eh] Void – System.Void	NULL	Method
7	0005h	[00001E44h] .ctor	[0005h] TestComponentWithMembers –	NULL	[001Eh] Void – System.Void	NULL	Constructor
8	0006h	[00001E44h] .ctor	[0006h] SampleAttributedComponent –	NULL	[001Eh] Void – System.Void	NULL	Constructor
9	0007h	[00001E44h] .ctor	[0007h] TestComponentHideRoot – Jet	NULL	[001Eh] Void – System.Void	NULL	Constructor
10	0008h	[00001E44h] .ctor	[0008h] TestComponentHidden – JetBr	NULL	[001Eh] Void – System.Void	NULL	Constructor
11	0009h	[00001E44h] .ctor	[0009h] TestComponentHiding – JetBr	NULL	[001Eh] Void – System.Void	NULL	Constructor
12	000Ah	[00001E44h] .ctor	[000Ah] TestComponentWithMultipleGe	NULL	[001Eh] Void – System.Void	NULL	Constructor
13	000Bh	[00001E44h] .ctor	[000Bh] ZoneMarker – JetBrains.Plat	NULL	[001Eh] Void – System.Void	NULL	Constructor
14	000Ch	[00001E44h] .ctor	[000Ch] TestComponentA – JetBrains.	NULL	[001Eh] Void – System.Void	0033h ∑1	Constructor
15	000Dh	[00001E44h] .ctor	[000Dh] TestComponentB – JetBrains.	NULL	[001Eh] Void – System.Void	0034h ∑1	Constructor
16	000Eh	[00001E44h] .ctor	[000Eh] TestComponentC – JetBrains.	NULL	[001Eh] Void – System.Void	0035h ∑2	Constructor
17	000Fh	[00001E44h] .ctor	[000Fh] TestComponentD – JetBrains.	NULL	[001Eh] Void – System.Void	0037h ∑1	Constructor
18	0010h	[00001E44h] .ctor	[0010h] TestComponentE – JetBrains.	NULL	[001Eh] Void – System.Void	0037h ∑1	Constructor
19	0011h	[00001E44h] .ctor	[0011h] TestComponentF – JetBrains.	NULL	[001Eh] Void – System.Void	0037h ∑1	Constructor
20	0012h	[00001E44h] .ctor	[0012h] TestComponentFromMember – J	NULL	[001Eh] Void – System.Void	0038h ∑1	Constructor
21	0013h	[00001E44h] .ctor	[0013h] TestComponentHieroImplement	NULL	[001Eh] Void – System.Void	NULL	Constructor
22	0014h	[00001E44h] .ctor	[0014h] TestComponentHieroImplement	NULL	[001Eh] Void – System.Void	NULL	Constructor
23	0015h	[00001E44h] .ctor	[0015h] TestComponentLifetimeClient	NULL	[001Eh] Void – System.Void	003Ch ∑1	Constructor
24	0016h	[00001E44h] .ctor	[0016h] TestComponentMulti1 – JetBr	NULL	[001Eh] Void – System.Void	NULL	Constructor
25	0017h	[00001E44h] .ctor	[0017h] TestComponentMulti2 – JetBr	NULL	[001Eh] Void – System.Void	NULL	Constructor
26	0018h	[00001E44h] .ctor	[0018h] TestComponentMultiClient –	NULL	[001Eh] Void – System.Void	003Eh ∑1	Constructor
27	0019h	[00003AA0h] MethodComponent	[0019h] TestComponentWithMembers –	0014h ∑1	[003Ah] IFactoriedFromMember – J	0040h ∑1	Method
28	001Ah	[00001E44h] .ctor	[0019h] TestComponentWithMembers –	NULL	[001Eh] Void – System.Void	003Fh ∑1	Constructor
29	001Bh	[00001E44h] .ctor	[001Ah] TestComponentPlugin – JetBr	NULL	[001Eh] Void – System.Void	NULL	Constructor

Managed API

- Без native pointers
- Со всеми бизнес-классами
  - `member.Type` должен вернуть объект `Type`
  - `member.Attributes` должен вернуть коллекцию
- Без memory traffic при простом обходе
  - Все эти объекты не должны нагружать GC

Delegating Envoys

- Value type
  - Все объекты в API — это структуры
  - Коллекции тоже структуры
  - C# позволяет `foreach` по структурам
    - Без боксинга в `IEnumerable`

Delegating Envoys

- Field #1: storage object
  - Один на всю структуру данных
  - Держит native память с BLOB’ами
  - Умеет извлекать из них данные
- Field #2: Token
  - Примитивного типа
  - Например, индекс записи в таблице таких объектов

Delegating Envoys

- Methods & Properties
  - Все полагающиеся по API методы и проперти
  - Делегируют вызов в такую же функцию из storage object
  - + дополнительный параметр: Token
  - Возвращает:
    - Примитивные типы
    - Delegating envoys
    - String envoys
    - Collection envoys

```
public struct PartCatalogTypeMember : IEquatable<PartCatalogTypeMember>
{
  private readonly IPartCatalogStorage myStorage;

  private readonly Int32<CatalogMemberToken> Token;

  public PartCatalogType DeclaringType
  {
    get
    {
      return myStorage.MemberGetDeclaringType(Token);
    }
  }

  public StringSource Name
  {
    get
    {
      return myStorage.MemberGetName(Token);
    }
  }
```

String Envoy

- Можно превратить в настоящий `string`
  - Тогда будет нагрузка на GC
- Нет нагрузки на GC до этого
  - Должен быть value type
- Все возможные строковые операции
  - Сравнение, хеш, подстрока, …
  - Между собой и со `string`
- Можно сделать из `string`

StringSource

- Value type
- Field #1: owning storage object
- Field #2: Arbitrary managed data (`Object`)
- Field #3: Arbitrary blittable data (12 bytes)
- Equality, Hash Code, Substring, StartsWith, etc
  - Может представлять runtime string, native memory, managed array

StringSource

```
public struct StringSource : IEquatable<StringSource>,
IComparable<StringSource>, IComparable
{
  private readonly IStringSourceOwner myOwner;
  public object DataRef;
  public StringSourcePodData DataPod;

  public bool IsEmpty { get { return Owner.IsEmpty(ref this); } }

  public char this[uint index]
  { get { return Owner.GetCharAt(ref this, index); } }

  public uint Length { get { return Owner.GetLength(ref this); } }

  public static bool operator ==(StringSource black, StringSource white)
  { return StringSourceComparerOrdinal.Equals(ref black, ref white); }

  public string ToRuntimeString()
  { return Owner.ToRuntimeString(ref this); }
```

StringSource Cons

- Ограниченное время жизни
  - После освобождения native memory невалиден
  - Все методы кидают exception
  - Можно неосторожно сохранить его в кеше
    - Функция AsManagedString() для этого
- Медленнее стандартной строки

Collection Envoy

- Нет нагрузки на GC при основных операциях:
  - Создание
  - C# `foreach`
  - `Count`
  - `Contains`
- Можно превратить в
  - `IEnumerable`1`
  - `ICollection`1`

CollectionSource

- Для списков объектов из таблиц
- Value type
- Field #1: owning storage object
- Field #2: Arbitrary managed data (`Object`)
- Field #3: Arbitrary blittable data (12 bytes)
  - Может представлять `ICollection`, `IEnumerable`, native memory, виртуальную коллекцию, вычислимый view

CollectionSource

- `GetEnumerator`: возвращает value type enumerator
- C# `foreach` в таком случае
  - Зовёт public `GetEnumerator` напрямую
  - Не делает boxing в `IEnumerable`1`
  - Использует value type enumerator напрямую
  - Не делает boxing в `IEnumerator`1`
- → `foreach` без memory traffic

CollectionSource cons

- Не реализует интерфейсы коллекций
  - Против случайного boxing
- Нет эффективного LINQ
  - Отдельные функции вроде `First()`, `Single()`
- Есть функции AsCollection, AsEnumerable, etc
  - Boxing

Value-type Envoys

- Load-time performance
  - Один managed storage object
    - На самом деле несколько, но O(1)
  - Нет обработки данных на загрузке
    - BLOB готов к чтению по запросу
- Runtime performance
  - Все объекты API — лёгкие value types
  - Возможна работа с данными без GC-аллокаций
    - Предварительная проверка и фильтрация строк и коллекций

Example 1

```
foreach(PartCatalog catalog in catalogset.Catalogs)
{
  foreach(PartCatalogType ctype in catalog.AllPartTypes)
  {
    foreach(PartCatalogTypeMember member in ctype.PartMembers)
    {
      if(!member.GetPartAttributes<PublicStaticIntMainAttribute>().IsEmpty())
        mains.Add(member);
    }
  }
}
```

Example 2

```
foreach(PartCatalogType part
                        in catalog.ApplyFilter(attributeFilter).AllPartTypes)
{
  foreach(PartCatalogAttribute attribute
                            in part.GetPartAttributes<TDefinitionAttribute>())
  {
    PartCatalogType? appliedDef =
                        attribute.ArgumentsOptional[myAttributeArgumentName]
                       .GetTypeValueIfDefined();

    myDefinitionsToParts.Add(catalog,
                           appliedDef == null ? null : appliedDef.Value.Bind());
  }
}
```

THE END