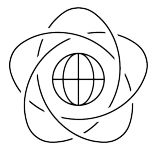
Programming 3 Advanced

Unsafe, C++ Interoperability

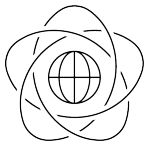
Tomasz Herman

Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 14, 19 stycznia 2025



Outline



Unsafe

```
<PropertyGroup>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

```
unsafe void BlueFilter (int[,] bitmap)
{
    fixed (int* b = bitmap)
    {
        int* p = b;
        for (int i = 0; i < bitmap.Length; i++)
            *p++ &= 0xFF;
    }
}
```

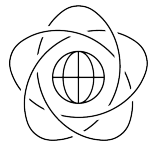
Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code

Pinning with fixed statement

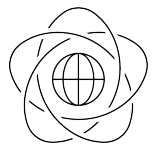
```
1  unsafe
2  {
3      int[] values = { 1, 2, 3 };
4      fixed (int* ptr = &values[0])
5      {
6          Console.WriteLine($"Value: {ptr[0]}");
7          Console.WriteLine($"Address: {(long)ptr:X}");
8      }
9  }
```

- Pinning prevents Garbage Collector from moving the object on the heap



Pointer operations

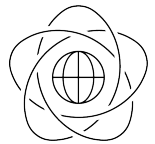
```
1  Test test = new Test();
2  unsafe
3  {
4      Test* p = &test;
5      p->X = 9;
6      int* x = &p->X;
7      *x = 10;
8      Console.WriteLine(test.X);
9  }
10 struct Test { public int X; }
```



stackalloc

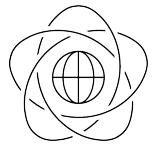
```
1 int* a = stackalloc int [10];
2 for (int i = 0; i < 10; ++i)
3     Console.WriteLine(a[i]);
```

- stackalloc can be also assigned to a Span<T>, which can be used in a safe context



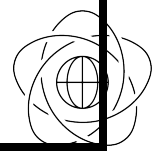
Fixed-size buffers

```
1  Console.WriteLine(sizeof(UnsafeStruct));
2  // Output: 48
3
4  unsafe struct UnsafeStruct
5  {
6      public long Offset;
7      public byte Length;
8      public fixed byte Buffer[32]; // 32 bytes directly
9                                    // in struct's memory
10 }
```



void*

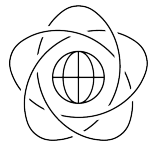
```
1  short[] arr = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
2  unsafe
3  {
4      fixed (short* ptr = arr)
5      {
6          Reset(ptr, arr.Length * sizeof(short));
7      }
8  }
9
10 foreach (short x in arr) Console.WriteLine(x);
11
12 unsafe void Reset(void* mem, int bytes)
13 {
14     byte* b = (byte*)mem;
15     for (int i = 0; i < bytes; i++)
16         *b++ = 0;
17 }
```



Function pointers

```
1  delegate*<int, char, string, void>
2      someFunctionPtr = &SomeFunction;
3  static void SomeFunction(int x, char y, string z) {}
4
5  delegate*<string, int> getLengthPtr = &GetLength;
6  int length = getLengthPtr("Hello, world");
7  static int GetLength (string s) => s.Length;
```

- Can only point to static functions

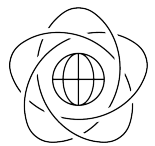


NativeMemory

.NET 6+

```
1  int* data = (int*)NativeMemory.Alloc(1024 * sizeof(int));
2
3  NativeMemory.Free(data);
```

- Thin wrapper for malloc/free
- Memory allocated that way is unmanaged by Garbage Collector
- Older applications might use `Marshal` class instead

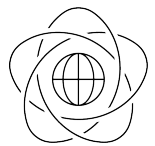


Types of Interoperability

- Native Interoperability
- COM Interoperability

Native Interoperability Guidelines

`https://learn.microsoft.com/en-us/dotnet/standard/native-interop/best-practices`



P/Invoke

Platform Invocation Services, allows an access to functions, structs, and callbacks in unmanaged libraries.

- DllImportAttribute
- LibraryImportAttribute

Documentation

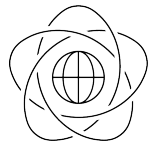
`https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke`

DllImport

```
#if defined(_WIN32) || defined(_WIN64)
#define EXPORT __declspec(dllexport)
#else
#define EXPORT __attribute__((visibility("default")))
#endif

extern "C" EXPORT void Hello();
```

```
[DllImport("HelloCpp")]
private static extern void Hello();
```



LibraryImport

```
#if defined(_WIN32) || defined(_WIN64)
#define EXPORT __declspec(dllexport)
#else
#define EXPORT __attribute__((visibility("default")))
#endif

extern "C" EXPORT void Hello();
```

```
[LibraryImport("HelloCpp")]
private static partial void Hello();
```

Differences from DllImport

https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke-source-generation#differences-from-dllimport

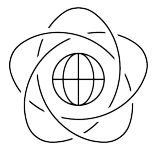
Marshalling Common Types

```
1 extern "C" EXPORT int Foo(unsigned char c, float x, Bar* bar);
```

```
1 [LibraryImport("HelloCpp")]
2 private static partial int Foo(byte c, float x, IntPtr bar);
```

Type Marshalling

https://learn.microsoft.com/en-us/dotnet/standard/native-interop/type-marshalling



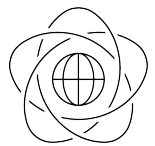
Marshalling Strings

As Parameters

```
1 extern "C" EXPORT void PrintAnsiString(const char* str);
2 extern "C" EXPORT void PrintUnicodeString(const char16_t* str);
```

```
1 [LibraryImport("StringsCpp",
2     StringMarshalling = StringMarshalling.Utf8)]
3 public static partial void PrintAnsiString(string str);
4 [LibraryImport("StringsCpp",
5     StringMarshalling = StringMarshalling.Utf16)]
6 public static partial void PrintUnicodeString(string str);
```

- It's also possible to marshall strings using MarshallAs attribute



Marshalling Strings

As Return Value

```
1 extern "C" EXPORT const char* GetAnsiString();
2 extern "C" EXPORT const char16_t* GetUnicodeString();
```

```
1 [LibraryImport("StringsCpp")]
2 public static partial IntPtr GetAnsiString();
3 [LibraryImport("StringsCpp")]
4 public static partial IntPtr GetUnicodeString();
5
6 string? ansi = Marshal.PtrToStringAnsi(GetAnsiString());
7 string? unicode = Marshal
8     .PtrToStringUni(GetUnicodeString());
```

String ownership

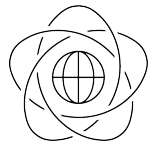
If necessary, it is the caller who is responsible to free resources associated with IntPtr.

Marshalling Strings

As In/Out Parameter

```
1  extern "C" EXPORT void Encode(char* text);
```

```
1  [DllImport("StringsCpp", CharSet = CharSet.Ansi)]
2  public static extern void Encode(StringBuilder str);
3
4  [LibraryImport("StringsCpp",
5      StringMarshalling = StringMarshalling.Utf8)]
6  public static partial void Encode(byte[] str);
```

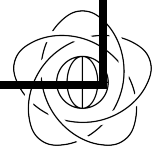


Marshalling Strings

In structs

```
1  struct StringsStruct
2  {
3      char *s1;
4      char  s2[256];
5  };
```

```
1  [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
2  struct StringInfoA
3  {
4      [MarshalAs(UnmanagedType.LPStr)]
5      public string s1;
6      [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)]
7      public string s2;
8  }
```



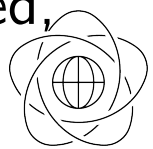
Callbacks from Unmanaged Code

Via Delegates

```
1  typedef void (*Callback)(int value);
2
3  extern "C" void Count(int from, int to, Callback callback);
```

```
1  public delegate void Callback(int value);
2
3  [LibraryImport("CallbacksCpp")]
4  private static partial void
5      Count(int from, int to, Callback callback);
6
7  Count(1, 101, i => {});
```

- When the delegate falls out of scope it might be Garbage Collected, causing runtime error when used by unmanaged code.



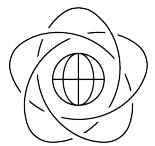
Callbacks from Unmanaged Code

Via function pointers

```
1  typedef void (*Callback)(int value);
2
3  extern "C" void Count(int from, int to, Callback callback);
```

```
1  [LibraryImport("CallbacksCpp")]
2  private static unsafe partial void
3      Count(int from, int to, delegate*<int, void> callback);
4
5  Count(1, 101, &FuncCallback);
6
7  static void FuncCallback(int i) {}
```

- With function pointers, the callback must be a static method

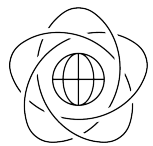


Callbacks from Unmanaged Code

Calling convention

```
1  [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
2  public delegate void Callback(int value);
3
4  [LibraryImport("CallbacksCpp")]
5  private static partial void
6      Count(int from, int to, Callback callback);
7
8  [LibraryImport("CallbacksCpp")]
9  private static unsafe partial voidCount(int from,
10     int to, delegate* unmanaged[Cdecl]<int, void> callback);
11
12 [UnmanagedCallersOnly(CallConvs = [typeof(CallConvCdecl)])]
13 static void FizzBuzz(int i)
```

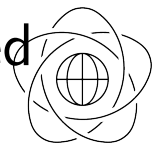
- By default, the compiler assumes that the unmanaged callback follows the platform default calling convention.



Marshalling a structure

```
1  [StructLayout(LayoutKind.Explicit)]
2  public struct Color
3  {
4      [FieldOffset(0)]
5      public byte R;
6      [FieldOffset(1)]
7      public byte G;
8      [FieldOffset(2)]
9      public byte B;
10     [FieldOffset(3)]
11     public byte A;
12     [FieldOffset(0)]
13     public uint Rgba;
14 }
```

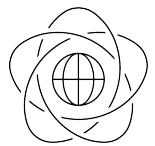
- The memory layout of the structure must match between managed and unmanaged code



Marshalling a structure

```
1 extern "C" EXPORT Color Add(Color a, Color b);
2 extern "C" EXPORT void Darken(Color *color);
3 extern "C" EXPORT void PrintHex(Color color);
```

```
1 [LibraryImport("StructsCpp")]
2 private static partial Color Add(Color a, Color b);
3
4 [LibraryImport("StructsCpp")]
5 private static partial void Darken(ref Color b);
6
7 [LibraryImport("StructsCpp")]
8 private static partial void PrintHex(Color a);
```

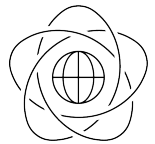


Safe Handles (1/2)

```
1  extern "C" EXPORT const char* CreateString();
2  extern "C" EXPORT void PrintString(const char* str);
3  extern "C" EXPORT void DestroyString(const char* str);
```

```
1  [LibraryImport("SafeHandleCpp")]
2  public static partial StringSafeHandle CreateString();
3
4  [LibraryImport("SafeHandleCpp")]
5  public static partial void PrintString(StringSafeHandle str);
6
7  [LibraryImport("SafeHandleCpp")]
8  public static partial void DestroyString(IntPtr str);
```

- SafeHandles allow automatic memory management through Dispose pattern.



Safe Handles (2/2)

```
1  using var str = NativeString.CreateString();
2  NativeString.PrintString(str);
3
4  public class StringSafeHandle : SafeHandle
5  {
6      public StringSafeHandle() : base(IntPtr.Zero, true) {}
7
8      protected override bool ReleaseHandle()
9      {
10         NativeString.DestroyString(handle);
11         handle = IntPtr.Zero;
12         return true;
13     }
14
15     public override bool IsInvalid => handle == IntPtr.Zero;
16 }
```