Interoperability between C# and other languages

Geza Kovacs

Managed vs Unmanaged Code

- Code you write in C# (or VB.NET, or F#, etc) is compiled into Common Intermediate Language (CIL) bytecode, which runs on the .NET framework (**managed code**)
- Code you write in C or C++ is compiled into machine code which runs directly on the machine (**unmanaged code**)

The Win32 API

- An unmanaged (written in C) library that allows your program to interact with Windows system services
- Ex: GetVersion() function in the Win32 API determines the version of Windows:

```
// from windows.h
DWORD WINAPI GetVersion();
```

The Win32 API

- An unmanaged (written in C) library that allows your program to interact with Windows system services
- Ex: GetVersion() function in the Win32 API determines the version of Windows:

```
// from windows.h
DWORD WINAPI GetVersion();
```


```
unsigned int __stdcall GetVersion();
```

The Win32 API

- An unmanaged (written in C) library that allows your program to interact with Windows system services
- Ex: GetVersion() function in the Win32 API determines the version of Windows:

```
// from windows.h
DWORD WINAPI GetVersion();
```

Equivalent to

```
unsigned int __stdcall GetVersion();
```

stdcall calling convention

P/Invoke (Platform Invoke)

- We can make functions from unmanaged code available to our C# program using P/Invoke

GetVersion() function is implemented in kernel32.dll

```
// from windows.h
DWORD WINAPI GetVersion();
```

Equivalent to

```
unsigned int __stdcall GetVersion();
```

```
[DllImport("kernel32.dll")]
static extern uint GetVersion();
```

P/Invoke (Platform Invoke)

- We can make functions from unmanaged code available to our C# program using P/Invoke

```
// from windows.h
DWORD WINAPI GetVersion();
```

Equivalent to

```
unsigned int __stdcall GetVersion();
```

```
[DllImport("kernel32.dll")]
static extern uint GetVersion();
```

"uint" in C# is equivalent to C datatype "unsigned int"

P/Invoke (Platform Invoke)

- We can make functions from unmanaged code available to our C# program using P/Invoke

```
// from windows.h
DWORD WINAPI GetVersion();
```


```
unsigned int __stdcall GetVersion();
```

```
using System;
using System.Runtime.InteropServices;
static class MyMainClass
{
    [DllImport("kernel32.dll")]
    static extern uint GetVersion();

    static void Main(string[] args)
    {
        uint winver = GetVersion();
    }
}
```

```
// from windows.h
DWORD WINAPI GetVersion();
```

Equivalent to

```
unsigned int __stdcall GetVersion();
```

```
using System;
using System.Runtime.InteropServices;
static class MyMainClass
{
    [DllImport("kernel32.dll")]
    static extern uint GetVersion();

    static void Main(string[] args)
    {
        uint winver = GetVersion();
        uint majorv = winver & 0xFF;
        uint minorv = (winver & 0xFF00) >> 8;
        Console.WriteLine("Windows version: " + majorv + "." + minorv);
    }
}
```

```
// from windows.h
DWORD WINAPI GetVersion();
```

Equivalent to

```
unsigned int __stdcall GetVersion();
```

If importing a function under a different name, use EntryPoint to specify the original function name

```
using System;
using System.Runtime.InteropServices;
static class MyMainClass
{
    [DllImport("kernel32.dll", EntryPoint="GetVersion")]
    static extern uint getver();

    static void Main(string[] args)
    {
        uint winver = getver();
        uint majorv = winver & 0xFF;
        uint minorv = (winver & 0xFF00) >> 8;
        Console.WriteLine("Windows version: " + majorv + "." + minorv);
    }
}
```

- For many APIs in Win32 like GetVersion(), .NET already provides a wrapper for you (for these you don't need to use P/Invoke)

```
using System;
static class MyMainClass
{
    static void Main(string[] args)
    {
        Version winver = Environment.OSVersion.Version;
        int majorv = winver.Major;
        int minorv = winver.Minor;
        Console.WriteLine("Windows version: " + majorv + "." + minorv);
    }
}
```

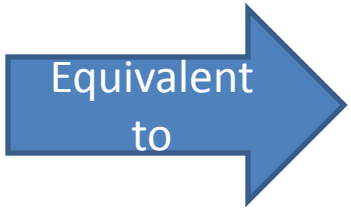
Calling Conventions

- Calling convention: convention for passing arguments to functions, and cleaning up the stack after the function has exited
- Libraries part of the Win32 API use the **stdcall** calling convention

```
int __stdcall someFunction(int arg);
```

- Most other libraries and C compilers, however, default to the **cdecl** calling convention

```
int __cdecl someFunction(int arg);
```




```
int someFunction(int arg);
```

Specifying calling convention

- Calling convention can be specified using the CallingConvention named argument
  - If not specified, P/Invoke defaults to stdcall

```
[DllImport("kernel32.dll")]
static extern uint GetVersion();
```


```
[DllImport("kernel32.dll", CallingConvention=CallingConvention.StdCall)]
static extern uint GetVersion();
```

- When using functions outside the Win32 API (ex: sqrt from the C standard library), **cdecl** calling convention should be specified

```
// from math.h
double sqrt(double x);
```

Equivalent to

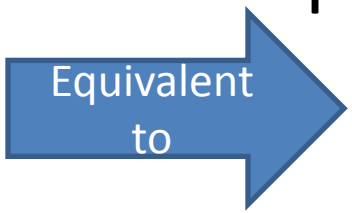
```
// from math.h
double __cdecl sqrt(double x);
```

C standard library is implemented in msvcrt.dll

```
[DllImport("msvcrt.dll", CallingConvention=CallingConvention.Cdecl)]
static double sqrt(double num);
```

- When using functions outside the Win32 API (ex: sqrt from the C standard library), **cdecl** calling convention should be specified

```
// from math.h
double sqrt(double x);
```



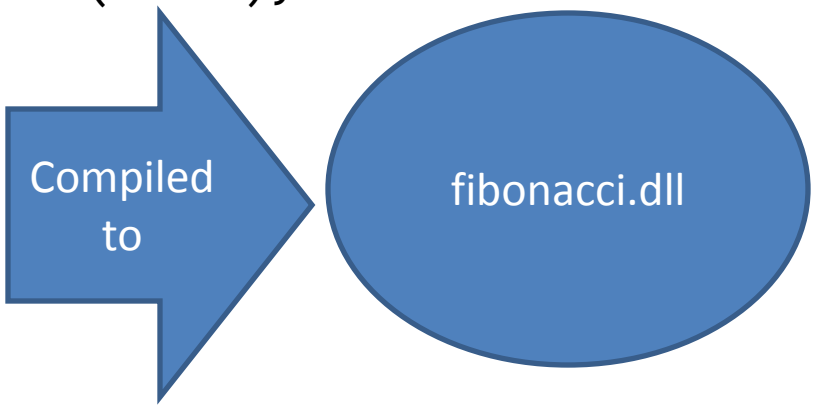

```
// from math.h
double __cdecl sqrt(double x);
```

```
using System;
using System.Runtime.InteropServices;

static class MyMainClass
{
  [DllImport("msvcrt.dll", CallingConvention=CallingConvention.Cdecl)]
  static double sqrt(double num);

  static void Main(string[] args)
  {
    double sqrtOfNine = sqrt(9.0);
    Console.WriteLine(sqrtOfNine);
  }
}
```

- C and C++ compilers also default to the **cdecl** calling convention for functions

```
// fib.h
extern "C" __declspec(dllexport) int fib(int n);
```

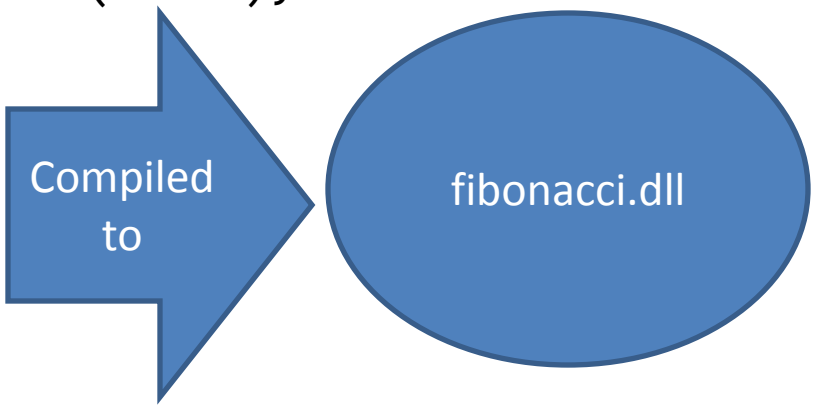
```
// fib.c
#include "fib.h"
int fib(int n) {
  if (n == 0 || n == 1) return 1;
  return fib(n-1) + fib(n-2);
}
```




```
[DllImport("fibonacci.dll", CallingConvention = CallingConvention.Cdecl)]
static extern int fib(int n);
```

- C and C++ compilers also default to the **cdecl** calling convention for functions

```
// fib.h
extern "C" __declspec(dllexport) int fib(int n);

// fib.c
#include "fib.h"
int fib(int n) {
  if (n == 0 || n == 1) return 1;
  return fib(n-1) + fib(n-2);
}
```




```
using System;
using System.Runtime.InteropServices;

static class MyMainClass {
  [DllImport("fibonacci.dll", CallingConvention = CallingConvention.Cdecl)]
  static extern int fib(int n);

  static void Main(string[] args) {
    Console.WriteLine(fib(8));
  }
}
```

Pointers

- An variable that refers to some location in memory
- Frequently used in C and C++, exists in C# mostly for interoperability purposes

```
int x = 5;
int* p;
p = &x;
Console.WriteLine(*p);
```

Pointers

- An variable that refers to some location in memory
- Frequently used in C and C++, exists in C# mostly for interoperability purposes

```
int x = 5;
int* p;
p = &x;
Console.WriteLine(*p);
```

Determines x's location in memory (address), makes p point to it

Pointers

- An variable that refers to some location in memory
- Frequently used in C and C++, exists in C# mostly for interoperability purposes

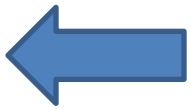
```
int x = 5;
int* p;
p = &x;
Console.WriteLine(*p);
```

dererences p (reads memory p points to)

Pointers

- Any location where pointers are used must be marked with the **unsafe** keyword

```
using System;

static class MyMainClass
{
    static void Main(string[] args)
    {
        unsafe
        {
            int x = 5;
            int* p;
            p = &x;
            Console.WriteLine(*p);
        }
    }
}
```



Pointers

- Any location where pointers are used must be marked with the **unsafe** keyword

```
using System;

static class MyMainClass
{
    static unsafe void Main(string[] args)
    {
        int x = 5;
        int* p;
        p = &x;
        Console.WriteLine((int)p);
    }
}
```

Pointers

- Any location where pointers are used must be marked with the **unsafe** keyword

```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int x = 5;
        int* p;
        p = &x;
        Console.WriteLine(*p);
    }
}
```

- Pointers can be passed to functions

```
using System;

static unsafe class MyMainClass
{
    static void swap(int* x, int* y)
    {
        int t = *y;
        *y = *x;
        *x = t;
    }

    static void Main(string[] args)
    {
        int q = 5;
        int r = 7;
        swap(&q, &r);
        Console.WriteLine(q); // 7
        Console.WriteLine(r); // 5
    }
}
```

- Pointers can refer to value types (like structs)

```
using System;

struct Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        Point* ptr = &p;
    }
}
```

- Pointers can refer to value types (like structs)

```
using System;

struct Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        Point* ptr = &p;
        (*ptr).x = 3;
        (*ptr).y = 5;
        Console.WriteLine(p.x); // 3
        Console.WriteLine(p.y); // 5
    }
}
```

- Pointers can refer to value types (like structs)

```
using System;

struct Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        Point* ptr = &p;
        ptr->x = 3;
        ptr->y = 5;
        Console.WriteLine(p.x); // 3
        Console.WriteLine(p.y); // 5
    }
}
```

ptr->x is a shorthand for (*ptr).x

- Pointers cannot refer to reference types (like classes)
  - Because the garbage collector might move class instances around

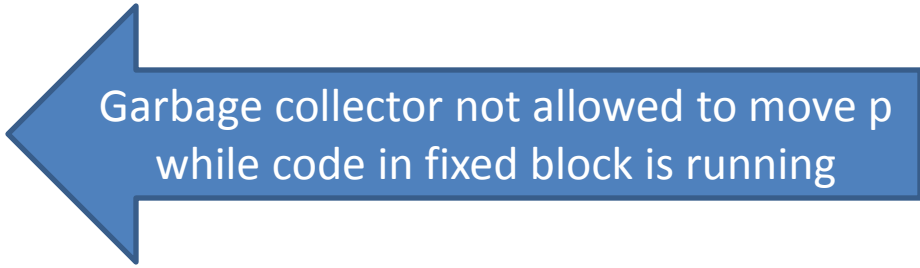
```
using System;

class Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        Point* ptr = &p; // NOT ALLOWED
    }
}
```

- Pointers can refer to fields in structs

```
using System;

struct Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        int* xPtr = &p.x;
        *xPtr = 5;
        Console.WriteLine(p.x);
    }
}
```

- If referring to a field in a class, use **fixed** to ensure the class instance doesn't get moved by the garbage collector

```
using System;

class Point
{
    public int x, y;
}

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point();
        fixed (int* xPtr = &p.x)
        {
            *xPtr = 5;
        }
        Console.WriteLine(p.x);
    }
}
```



- If using pointers to elements of an array, also need to use **fixed**

```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int[] arr = new int[10];
        fixed (int* p = &arr[0])
        {
            *p = 5;
        }
    }
}
```

- If using pointers to elements of an array, also need to use **fixed**

```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int[] arr = new int[10];
        fixed (int* p = &arr[0])
        {
            *p = 5;
        }
    }
}
```

- Pointer arithmetic is allowed in C#

```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int[] arr = new int[10];
        fixed (int* p = &arr[0])
        {
            *p = 5;
            *(p + 1) = 6;
        }
    }
}
```

- Pointer arithmetic is allowed in C#

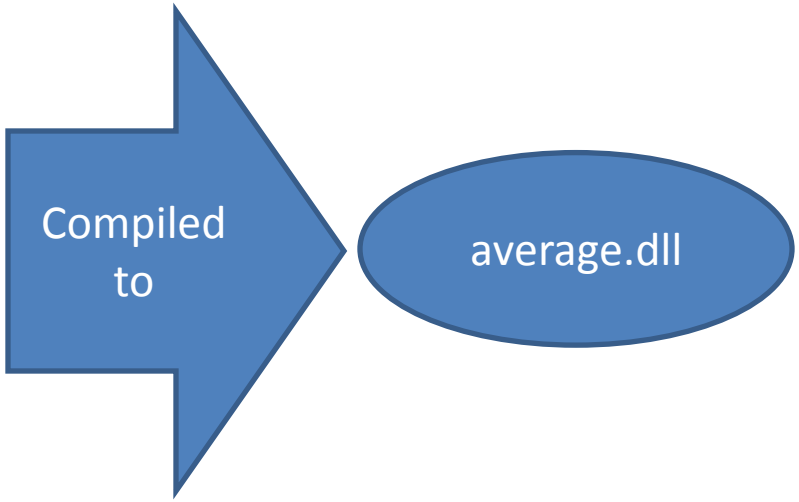
```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int[] arr = new int[10];
        fixed (int* p = &arr[0])
        {
            *p = 5;
            *(p + 1) = 6;
            *(p + 2) = 8;
        }
    }
}
```

- Pointer arithmetic is allowed in C#
  - Indexing syntax can also be used: p[i] is equivalent to *(p+i)

```
using System;

static unsafe class MyMainClass
{
    static void Main(string[] args)
    {
        int[] arr = new int[10];
        fixed (int* p = &arr[0])
        {
            *p = 5;
            *(p + 1) = 6;
            p[2] = 8;
        }
    }
}
```

```
// average.h
extern "C" __declspec(dllexport) double average(double *list, int length);
```

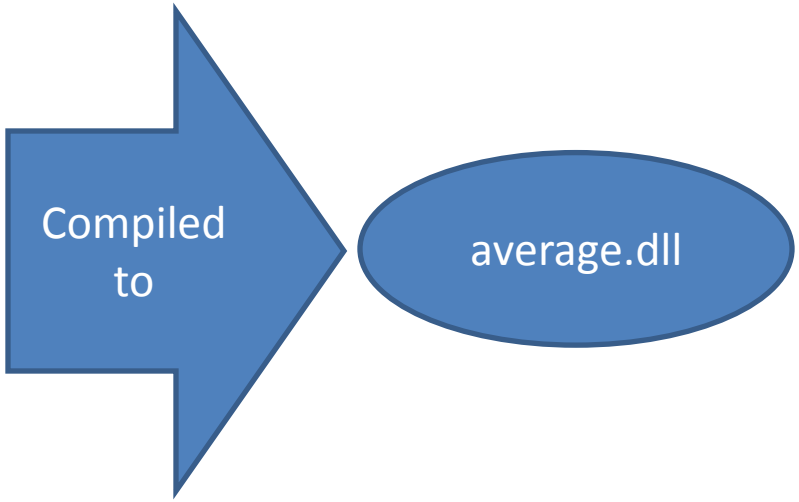
```
// average.c
#include "average.h"

double average(double *list, int length) {
  double total = 0.0;
  for (int i = 0; i < length; ++i)
    total += list[i];
  return total / length;
}
```




```
[DllImport("average.dll", CallingConvention = CallingConvention.Cdecl)]
static extern double average(double* list, int length);
```

```h
// average.h
extern "C" __declspec(dllexport) double average(double *list, int length);
```

```c
// average.c
#include "average.h"

double average(double *list, int length) {
  double total = 0.0;
  for (int i = 0; i < length; ++i)
    total += list[i];
  return total / length;
}
```




```cs
using System;
using System.Runtime.InteropServices;
static unsafe class MyMainClass {
    [DllImport("average.dll", CallingConvention = CallingConvention.Cdecl)]
    static extern double average(double* list, int length);

    static void Main(string[] args) {
        double[] arr = new double[] { 2,4,6,8,9,4,6,6,8,6 };
        fixed (double* p = &arr[0]) {
            Console.WriteLine(average(p, arr.Length));
        }
    }
}
```

- C-style string: character array terminated by 0
- puts: part of the C standard library, prints a C-style string
- Note: char in C is 1 byte, char in C# is 2 bytes. Use C# sbyte datatype instead.

```
// from stdio.h
int __cdecl puts(char* str);
```

```
[DllImport("msvcrt.dll", CallingConvention = CallingConvention.Cdecl)]
static extern void puts(sbyte* str);
```

- C-style string: character array terminated by 0
- puts: part of the C standard library, prints a C-style string
- Note: char in C is 1 byte, char in C# is 2 bytes. Use C# sbyte datatype instead.

```
// from stdio.h
int __cdecl puts(char* str);
```

```
using System;
using System.Runtime.InteropServices;

static unsafe class MyMainClass {
  [DllImport("msvcrt.dll", CallingConvention = CallingConvention.Cdecl)]
  static extern void puts(sbyte* str);

  static void Main(string[] args) {
    sbyte[] msg = new sbyte[] { (sbyte)'h', (sbyte)'i', (sbyte)'!', 0};
    fixed (sbyte* p = &msg[0]) {
      puts(p);
    }
  }
}
```

- GetUserName(): Win32 API function that writes the username into the given character buffer

```
BOOL WINAPI GetUserName(LPTSTR lpBuffer, LPDWORD lpnSize);
```

equivalent to

```
bool __stdcall GetUserName(char* buffer, unsigned int* bufferSize);
```

```
[DllImport("advapi32.dll")]
static extern bool GetUserName (sbyte* buffer, uint* bufferSize);
```

- GetUserName(): Win32 API function that writes the username into the given character buffer

```
BOOL WINAPI GetUserName(LPTSTR lpBuffer, LPDWORD lpnSize);
```

equivalent to

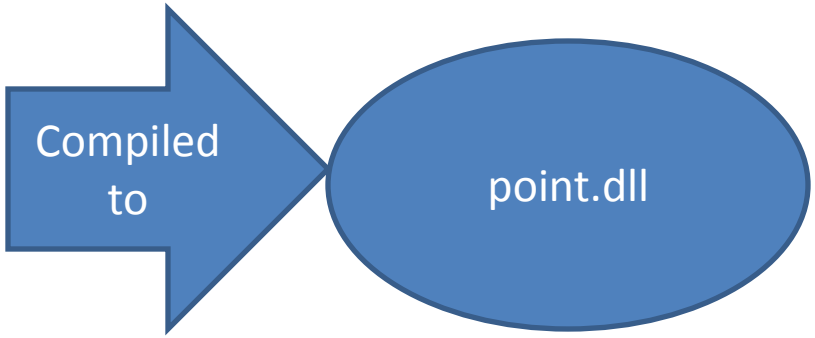
```
bool __stdcall GetUserName(char* buffer, unsigned int* bufferSize);
```

```
using System;
using System.Runtime.InteropServices;
static unsafe class MyMainClass {
    [DllImport("advapi32.dll")]
    static extern bool GetUserName(sbyte* buffer, uint* bufferSize);

    static void Main(string[] args) {
        sbyte[] buf = new sbyte[1024];
        uint size = 1024;
        string name;
        fixed (sbyte* p = &buf[0]) {
            GetUserName(p, &size);
            name = new string(p);
        }
        Console.WriteLine(name);
    }
}
```

```
// point.h
struct Point { int x, y; };
extern "C" __declspec(dllexport) Point addPoints(Point a, Point b);
```

a struct in C, with fields x and y

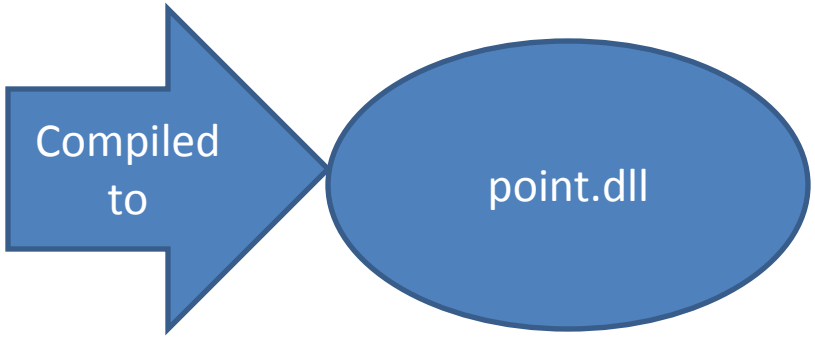
```
// point.cpp
#include "point.h"
Point addPoints(Point a, Point b) {
    Point c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}
```



- Can also pass custom C and C++ datatypes (struct, class) using P/Invoke

```
// point.h
struct Point { int x, y; };
extern "C" __declspec(dllexport) Point addPoints(Point a, Point b);
```

```
// point.cpp
#include "point.h"
Point addPoints(Point a, Point b) {
    Point c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}
```



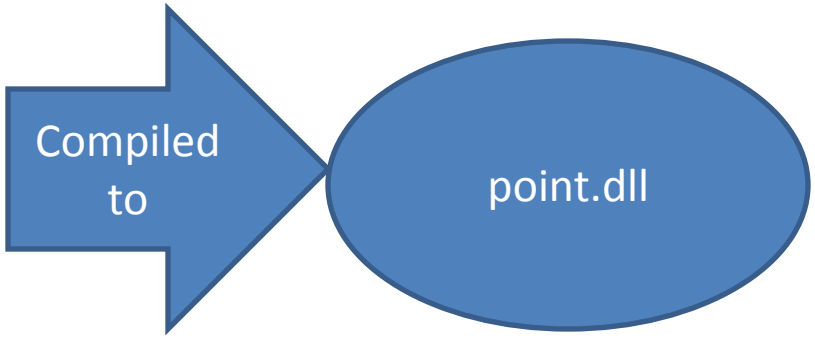

```
using System;
using System.Runtime.InteropServices;

struct Point { public int x, y; }

static class MyMainClass {
    [DllImport("point.dll", CallingConvention=CallingConvention.Cdecl)]
    static extern Point addPoints(Point a, Point b);

    static void Main(string[] args) {
        Point a; Point b;
        a.x = 4; a.y = 7; b.x = 3; b.y = 9;
        Point c = addPoints(a, b);
    }
}
```

Need to define a C# struct with same fields

```
// point.h
class Point { public: int x, y; };
extern "C" __declspec(dllexport) Point addPoints(Point a, Point b);
```

a class in C++, with fields x and y

```
// point.cpp
#include "point.h"
Point addPoints(Point a, Point b) {
    Point c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}
```




```
using System;
using System.Runtime.InteropServices;

struct Point { public int x, y; }

static class MyMainClass {
    [DllImport("point.dll", CallingConvention=CallingConvention.Cdecl)]
    static extern Point addPoints(Point a, Point b);

    static void Main(string[] args) {
        Point a; Point b;
        a.x = 4; a.y = 7; b.x = 3; b.y = 9;
        Point c = addPoints(a, b);
    }
}
```

Pass C++ classes as C# structs, not classes

C++/CLI

- Often, want to use both libraries written in both managed (C#, VB.NET, F#) and unmanaged code (C, C++)
- P/Invoke with C# is suboptimal – requires rewriting each signature for each unmanaged function, and rewriting all fields in each unmanaged type
- Solution: use C++ with a set of extensions allowing managed code to be invoked (C++/CLI)

Hello World in C# vs C++/CLI

- In C#, everything must be in a class. In C++/CLI, functions outside classes are also allowed

```
using System;

static class MyMainClass {
    static void Main() {
        Console.WriteLine("hello world");
    }
}
```

```
using namespace System;

int main()
{
    Console::WriteLine("hello world");
}
```

Static methods called with ::

```
using System;
using System.Collections.Generic;

static class MyMainClass {
    static void Main() {
        LinkedList<int> list = new LinkedList<int>();
        for (int i = 0; i < 10; ++i)
            list.AddLast(i);
        foreach (int i in list)
            Console.WriteLine(i);
    }
}
```

```
#using "System.dll"
```

Reference are listed within the C++/CLI file

```
using namespace System;
using namespace System::Collections::Generic;

int main() {
    LinkedList<int> ^list = gcnew LinkedList<int>();
    for (int i = 0; i < 10; ++i)
        list->AddLast(i);
    for each (int i in list)
        Console::WriteLine(i);
}
```

```
using System;
using System.Collections.Generic;

static class MyMainClass {
    static void Main() {
        LinkedList<int> list = new LinkedList<int>();
        for (int i = 0; i < 10; ++i)
            list.AddLast(i);
        foreach (int i in list)
            Console.WriteLine(i);
    }
}
```

```
#using "System.dll"

using namespace System;
using namespace System::Collections::Generic;

int main() {
    LinkedList<int> ^list = gcnew LinkedList<int>();
    for (int i = 0; i < 10; ++i)
        list->AddLast(i);
    for each (int i in list)
        Console::WriteLine(i);
}
```

```
using System;
using System.Collections.Generic;

static class MyMainClass {
    static void Main() {
        LinkedList<int> list = new LinkedList<int>();
        for (int i = 0; i < 10; ++i)
            list.AddLast(i);
        foreach (int i in list)
            Console.WriteLine(i);
    }
}
```

```
#using "System.dll"

using namespace System;
using namespace System::Collections::Generic;

int main() {
    LinkedList<int> ^list = gcnew LinkedList<int>();
    for (int i = 0; i < 10; ++i)
        list->AddLast(i);
    for each (int i in list)
        Console::WriteLine(i);
}
```

^ is pointer to managed type

```
using System;
using System.Collections.Generic;

static class MyMainClass {
    static void Main() {
        LinkedList<int> list = new LinkedList<int>();
        for (int i = 0; i < 10; ++i)
            list.AddLast(i);
        foreach (int i in list)
            Console.WriteLine(i);
    }
}
```

```
#using "System.dll"

using namespace System;
using namespace System::Collections::Generic;

int main() {
    LinkedList<int> ^list = gcnew LinkedList<int>();
    for (int i = 0; i < 10; ++i)
        list->AddLast(i);
    for each (int i in list)
        Console::WriteLine(i);
}
```

gcnew used to allocate managed (garbage-collected) types

```
using System;
using System.Collections.Generic;

static class MyMainClass {
    static void Main() {
        LinkedList<int> list = new LinkedList<int>();
        for (int i = 0; i < 10; ++i)
            list.AddLast(i);
        foreach (int i in list)
            Console.WriteLine(i);
    }
}
```

```
#using "System.dll"

using namespace System;
using namespace System::Collections::Generic;

int main() {
    LinkedList<int> ^list = gcnew LinkedList<int>();
    for (int i = 0; i < 10; ++i)
        list->AddLast(i);
    for each (int i in list)
        Console::WriteLine(i);
}
```

-> used to invoke instance methods

:: used to invoke static methods

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass {
    static void Main() {
        Point p = new Point(3, 7);
    }
}
```

```
ref class Point {
public:
    int x;
    int y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point ^p = gcnew Point(3, 7);
}
```

- Access modifier in C++/CLI is denoted using **public:** (or **private:**, etc); applies to all following fields and methods

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass {
    static void Main() {
        Point p = new Point(3, 7);
    }
}
```

```
ref class Point {
public:
    int x;
    int y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point ^p = gcnew Point(3, 7);
}
```

- **ref class** is a reference type; corresponds to "class" in C#
- default access modifier is **private**

	Default private	Default public
Reference type	**ref class**	ref struct
Value type	value class	value struct
Unmanaged type	class	struct

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass {
    static void Main() {
        Point p = new Point(3, 7);
    }
}
```

```
ref struct Point {
    int x;
    int y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point ^p = gcnew Point(3, 7);
}
```

- **ref struct** is also a reference type; also corresponds to "class" in C#
- default access modifier is **public**

	Default private	Default public
Reference type	ref class	**ref struct**
Value type	value class	value struct
Unmanaged type	class	struct

```
struct Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass {
    static void Main() {
        Point p = new Point(3, 7);
    }
}
```

```
value class Point
public:
    int x;
    int y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point ^p = gcnew Point(3, 7);
}
```

- **value class** is a value type; corresponds to "struct" in C#
- default access modifier is **private**

	Default private	Default public
Reference type	ref class	ref struct
Value type	**value class**	value struct
Unmanaged type	class	struct

```
struct Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass {
    static void Main() {
        Point p = new Point(3, 7);
    }
}
```

```
value struct Point {
    int x;
    int y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point ^p = gcnew Point(3, 7);
}
```

- **value struct** is a value type; corresponds to “struct” in C#
- default access modifier is **public**

	Default private	Default public
Reference type	ref class	ref struct
Value type	value class	**value struct**
Unmanaged type	class	struct

```
class Point {
public:
    int x, y;
};
```

Cannot be “ref class” or “value class”

```
class Vector {
public:
    Point start, end;
};
```

- A unmanaged class cannot be the field of a managed class

```
value class Point {
public:
    int x, y;
};
```

can't be "class"

```
value class Vector {
public:
    Point start, end;
};
```

- A unmanaged type cannot be the field of a managed class
- A managed type cannot be the field of an unmanaged class

```
#include <vcclr.h>

ref class Point {
public:
    int x, y;
};

class Vector {
public:
    gcroot<Point^> start;
    gcroot<Point^> end;
    Vector() {
        start = gcnew Point;
        end = gcnew Point;
    }
};

int main() {
    Vector v;
    v.start->x = 5;
    v.start->y = 6;
}
```

- **gcroot<T>** can be used to wrap ref and value types as unmanaged types

```
class Point {
public:
    int x, y;
};

ref class Vector {
public:
    Point *start;
    Point *end;
    Vector() {
        start = new Point;
        end = new Point;
    }
    ~Vector() {
        delete start;
        delete end;
    }
};

int main() {
    Vector^ v = gcnew Vector;
    v->start->x = 5;
    v->start->y = 6;
}
```

- **gcroot<T>** can be used to wrap ref and value types as unmanaged types
- Ref and value classes can have pointers to unmanaged classes
  - Destructor: gets run when managed type becomes inaccessible

- Managed classes should be stored in managed containers (array<T>, System::Collections::Generic::LinkedList<T>)
- Unmanaged classes should be stored in unmanaged containers (standard array, std::list<T>)

```
ref class ManagedPoint {
public:
    int x, y;
};

class Point {
public:
    int x, y;
};

int main() {
    Point* arr = new Point[8];
    delete[] arr;
    array<ManagedPoint^>^ marr = gcnew array<ManagedPoint^>(8);
}
```

- Managed classes should be stored in managed containers (array<T>, System::Collections::Generic::LinkedList<T>)
- Unmanaged classes should be stored in unmanaged containers (standard array, std::list<T>)

```
#using "System.dll"
#include <list>
using namespace System::Collections::Generic;

ref class ManagedPoint {
public:
    int x, y;
};

class Point {
public:
    int x, y;
};

int main() {
    std::list<Point> lst;
    LinkedList<ManagedPoint^>^ mlst = gcnew LinkedList<ManagedPoint^>();
}
```

- Additionally, STL/CLR, an implementation of the STL for managed types, is available in the **cliext** namespace

```
#using "Microsoft.VisualC.STLCLR.dll"

#include <cliext/list>

ref class ManagedPoint {
public:
    int x, y;
};

int main() {
    ManagedPoint ^p = gcnew ManagedPoint;
    p->x = 5; p-> y = 7;
    cliext::list<ManagedPoint^> ^lst = gcnew cliext::list<ManagedPoint^>;
    lst->push_back(p);
}
```

```
class Point {
public:
    int x, y;
};

ref class ManagedPoint {
public:
    int x, y;
};

ManagedPoint^ addPoints(Point a, ManagedPoint ^b) {
    ManagedPoint ^c = gcnew ManagedPoint();
    c->x = a.x + b->x;
    c->y = a.y + b->y;
    return c;
}

int main() {
    Point p;
    p.x = 3; p.y = 5;
    ManagedPoint ^m = gcnew ManagedPoint();
    m->x = 5; m-> y = 8;
    ManagedPoint ^q = addPoints(p, m);
}
```

- Functions and methods can have both managed and unmanaged types as arguments

```
class Point {
public:
    int x, y;
};

ref class ManagedPoint {
public:
    int x, y;
    Point addPoint(Point a) {
        Point p;
        p.x = a.x + x;
        p.y = a.x + y;
        return p;
    }
};

int main() {
    Point p; p.x = 3; p.y = 5;
    ManagedPoint ^m = gcnew ManagedPoint();
    m->x = 5; m-> y = 8;
    Point q = m->addPoint(p);
}
```

- Functions and methods can have both managed and unmanaged types as arguments

- Functions and methods can have both managed and unmanaged types as arguments

```
ref class ManagedPoint {
public:
    int x, y;
};

class Point {
public:
    int x, y;
    ManagedPoint^ addPoint(ManagedPoint ^a) {
        ManagedPoint ^p = gcnew ManagedPoint;
        p->x = a->x + x;
        p->y = a->y + y;
        return p;
    }
};

int main() {
    Point p; p.x = 3; p.y = 5;
    ManagedPoint ^m = gcnew ManagedPoint();
    m->x = 5; m-> y = 8;
    ManagedPoint ^q = p.addPoint(m);
}
```

```
using System;

static class MyMainClass {
    static int square(int x) {
        return x * x;
    }
    static void Main() {
        Func<int, int> sq = square;
        Console.WriteLine(sq(4));
    }
}
```

- Delegates are available; can reference:
  - **functions**
  - static methods
  - instance methods

```
using namespace System;

int square(int x) {
    return x * x;
}

int main() {
    Func<int, int>^ sq = gcnew Func<int, int>(square);
    Console::WriteLine(sq(4));
}
```

```
using System;

static class MyMainClass {
     static int square(int x) {
          return x * x;
     }
     static void Main() {
          Func<int, int> sq = square;
          Console.WriteLine(sq(4));
     }
}
```

- Delegates are available; can reference:
  - functions
  - **static methods**
  - instance methods

```
using namespace System;

ref class Utils {
public:
    static int square(int x) {
        return x*x;
    }
};

int main() {
    Func<int, int> ^sq = gcnew Func<int, int>(Utils::square);
    Console::WriteLine(sq(4));
}
```

```
using System;
using System.Collections.Generic;

static class MyMainClass {
  static void Main() {
    HashSet<int> set = new HashSet<int>();
    set.Add(5);
    set.Add(7);
    Func<int, bool> contains = set.Contains;
    Console.WriteLine(contains(7)); // True
  }
}
```

- Delegates are available; can reference:
  - functions
  - static methods
  - **instance methods**

```
#using "System.dll"
#using "System.Core.dll"

using namespace System;
using namespace System::Collections::Generic;

int main() {
  HashSet<int>^ set = gcnew HashSet<int>();
  set->Add(5);
  set->Add(7);
  Func<int, bool>^ contains = gcnew Func<int, bool>(set, &HashSet<int>::Contains);
  Console::WriteLine(contains(7)); // True
}
```

Final notes on C++/CLI

- C++/CLI also lacks many other syntactic features of C#:
  - Type inference (var)
  - Lambdas
  - Extension methods (need to pass in **this** argument manually)
  - LINQ (since it consists of extension methods)
- Hence, generally C++/CLI is used only for wrapping libraries into a managed class to be used in C#

Interop with Java

- The JVM is similar to the CLR: both are virtual machines on which platform-independent bytecode runs
- Compiled Java bytecode (.jar files) can be converted into .NET bytecode (assemblies) using IKVM http://www.ikvm.net/
- Using IKVM, you can create and use instances of classes defined in Java code within your C# program