unsafe, fixed, P/Invoke

Карасева Елизавета Олеговна
liza.1610@mail.ru

07.09.2022

Введение

- C# безопаснее, чем C или C++
  - Вместо указателей — ссылки и объекты
  - Всё отслеживается сборщиком мусора
  - Нет ошибок с инициализацией или висячими указателями
- Небезопасный код
  - Работа с ОС или устройством, отображенным в памяти
  - Можно объявлять указатели, обращаться к переменным по адресу, выделять блоки памяти
  - Иногда увеличивает скорость работы
  - Риски для стабильности и безопасности

unsafe

- Небезопасный контекст
  - Вызов функций, требующих указатели
  - Пишется в объявлении методов:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count )
{
    // Unsafe context: can use pointers here.
}
```

  - Можно использовать блок unsafe{ *тут написан код* }
- Для компиляции задаём параметр:

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

Типы и объявление указателей

- Тип указателя — ссылочный
  - enum, указатель, byte, int, char, float, bool, ...

```
type* identifier;
void* identifier; //allowed but not recommended
```

- Не наследуется и не преобразуется в object
- Можно преобразовывать в целочисленные
- Данные не отслеживаются сборщиком мусора

- int* p1, p2, p3; // Ok
- int *p1, *p2, *p3; // Invalid in C#

Работа с указателями

- Указывает на наименьший адресуемый байт переменной
- * используется для доступа к содержимому
- void *
  - Никакой арифметики и косвенного обращения
  - Можно привести к другому типу указателя

- Конструкции для работы:
  - ->
  - &
  - stackalloc
    - System.Span<T> или System.ReadOnlySpan<T>
  - fixed
  - in, out или ref

Пример

```
int number = 1024;
unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;
    System.Console.Write("The 4 bytes of the integer:");
    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);
    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
     */
}
```

fixed

- Фиксирует переменную и объявляет указатель
- Адрес readonly и действителен только в блоке fixed

- Инициализация:
  - Первый элемент массива или строки
  - Переменная, поля объектов
    - Экземпляр объекта не будет перемещен или удален
  - Тип, реализующий метод GetPinnableReference
    - System.Span<T> .NET и System.ReadOnlySpan<T>
  - Буфер фиксированного размера

Буфер фиксированного размера

- Только в unsafe контексте
- Только поле экземпляра структуры
- Всегда векторы или одномерные массивы
- Объявление должно включать длину:
  - fixed char id[8]; // Ok
  - fixed char id[]; // Invalid in C#

Пример

```
internal unsafe struct Buffer{ public fixed char fixedBuffer[128]; }
internal unsafe class Example { public Buffer buffer = default; }
private static void AccessEmbeddedArray()
{
    var example = new Example();
    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);
        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

P/Invoke

- Управляет ОС
- Функций из неуправляемых библиотек в управляемом коде
- Обратно — через делегаты
  - Управляемый код — под управлением CLR
  - Неуправляемый — вне среды выполнения CLR
  - Например, функции Win32 API, компоненты COM, ...
- static extern метод с атрибутом DllImport
- System и System.Runtime.InteropServices

DllImport

- Объявление функции
- Сообщает компилятору, где находится точка входа

```
[System.AttributeUsage(System.AttributeTargets.Method, Inherited=false)]
public sealed class DllImportAttribute : Attribute
```

- Для использования экспортированных фунций:
  1. Определяем функцию в DLL
  2. Создаем класс для хранения функций DLL
  3. Создаем прототипы в управляемом коде
  4. Вызываем функцию DLL

Вызываем неуправляемый код из управляемого

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

public class Program
{
   // Import user32.dll (containing the function we need) and define
   // the method corresponding to the native function.
   [DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
   private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption,
   uint uType);

   public static void Main(string[] args)
   {
      // Invoke the function as a regular managed method.
      MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
   }
}
```

Вызываем неуправляемый код из управляемого

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

namespace ConsoleApplication1
{
   public static class Program
   {
      // Define a delegate that corresponds to the unmanaged function.
      private delegate bool EnumWC(IntPtr hwnd, IntPtr lParam);

      // Import user32.dll (containing the function we need) and define
      // the method corresponding to the native function.
      [DllImport("user32.dll")]
      private static extern int EnumWindows(EnumWC lpEnumFunc, IntPtr lParam);

      // Define the implementation of the delegate; here, we simply output the window handle.
      private static bool OutputWindow(IntPtr hwnd, IntPtr lParam)
      {
         Console.WriteLine(hwnd.ToInt64());
         return true;
      }

      public static void Main(string[] args)
      {
         // Invoke the method; note the delegate as a first parameter.
         EnumWindows(OutputWindow, IntPtr.Zero);
      }
   }
}
```

Маршалинг

- Преобразование типов при переходе от управляемого кода к машинному
- Необходимость — различие типов
- Разные случаи:
  - default marshaling type (для строк — LPTSTR)
  - Функции с выходным строковым параметром char*
  - MarshalAs
  - Функций, требующие struct
  - Функции обратного вызова
  - UnmanagedType.CustomMarshaler и ICustomMarshaler для пользовательского маршалинга

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

Структура в управляемом коде

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime systemTime);
[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milsecond;
}
public static void Main(string[] args) {
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

Та же структура в неуправляемом коде

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Процесс маршалинга

- Уплавление памятью:
  - Среда выполнения выделяет часть неуправляемой памяти
  - Данные управляемого класса копируются
  - Вызывается неуправляемая функция
  - Неуправляемая память копируется обратно в управляемую
- Неявное управление памятью
- Контроль времени жизни функций

Полезные ссылки

- Документация по unsafe-коду от Microsoft
- Пример, где fixed действительно ускоряет программу
- DllImport
- Замечательная статья про DllImport и маршалинг
- Подробнее про маршалинг разных типов