

Maximizing C# Performance: Advanced Coding Patterns for High-Efficiency Apps

Dr. Otto Dobretsberger
Sr. Software Dev

About Otto

Sr. Software Dev @ CODE Consulting

Professor at UH

2x World Records in Benchpress



otto@codemag.com



30 years of

"Helping People Build Better Software"

AI Consulting Services

CODE CONSULTING

Check out our new Executive Briefing offer!

We can help with your AI needs

What does AI mean for you?

“Skunk Works” Projects

codemag.com/AI
codemag.com/ExecutiveBriefing



Agenda



1. Introduction
2. Reducing Loop Cascades
3. Span Slicing vs. Substrings
4. Array Pooling
5. Zero Copy Techniques
6. Structs of Arrays
7. Stackalloc
8. LINQ Avoidance
9. Garbage Collection Optimization



Introduction

Introduction

- Overview of High-Performance Coding
- Outline of Key Topics

- Importance of Optimization in Blazor Applications





Reducing Loop Cascades

Reducing Loop Cascades

- Common Pitfalls in Nested Loops
- Strategies for Flattening Loops
- Examples

Pitfalls

- Increased Time Complexity
- Memory Overhead
- Cache Misses

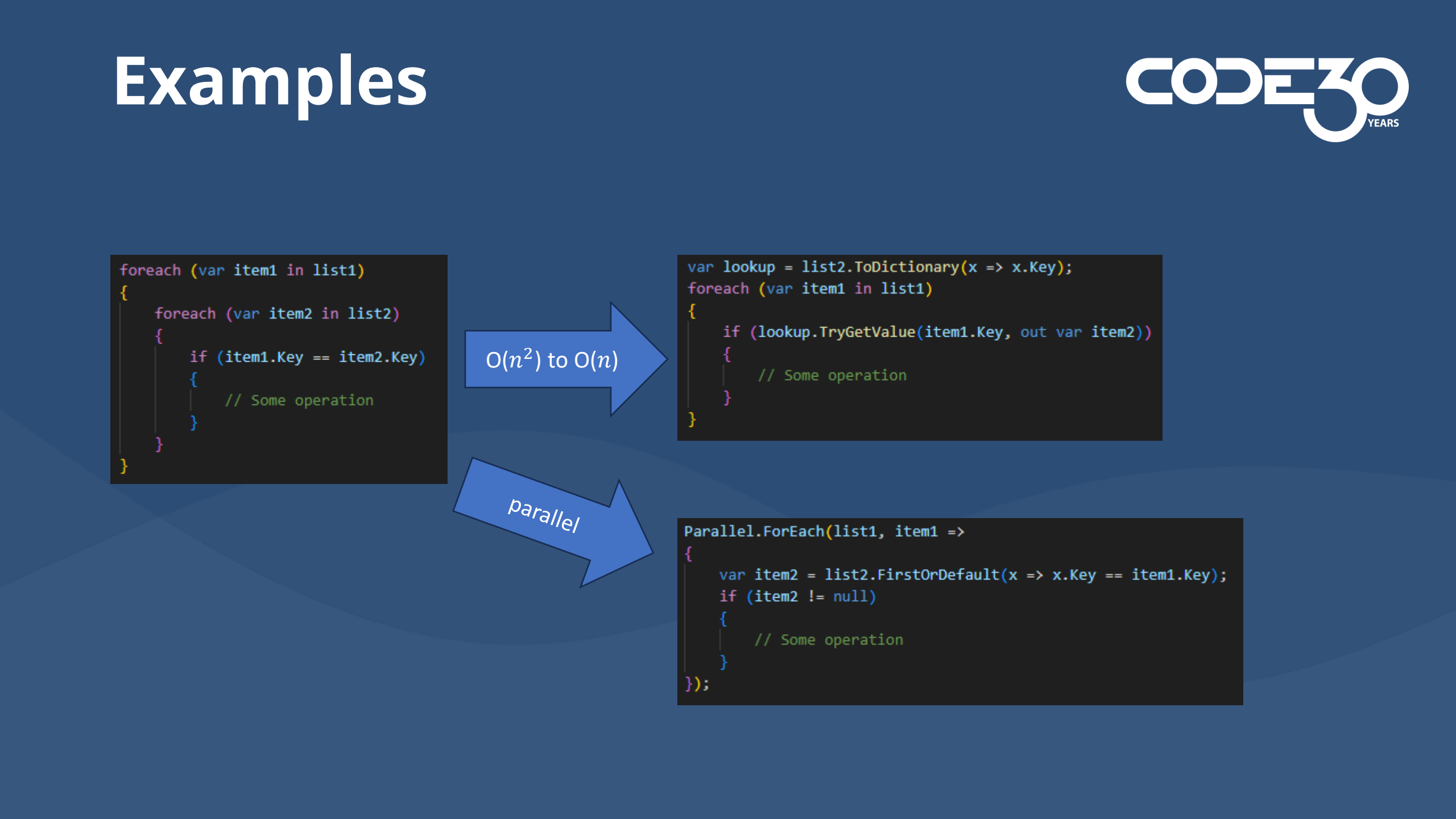


Strategies

- Break down the problem
- Use efficient Data Structures
- Leverage Parallelism



Examples




```
foreach (var item1 in list1)
{
    foreach (var item2 in list2)
    {
        if (item1.Key == item2.Key)
        {
            // Some operation
        }
    }
}
```

$O(n^2)$ to $O(n)$

```
var lookup = list2.ToDictionary(x => x.Key);
foreach (var item1 in list1)
{
    if (lookup.TryGetValue(item1.Key, out var item2))
    {
        // Some operation
    }
}
```

parallel

```
Parallel.ForEach(list1, item1 =>
{
    var item2 = list2.FirstOrDefault(x => x.Key == item1.Key);
    if (item2 != null)
    {
        // Some operation
    }
});
```

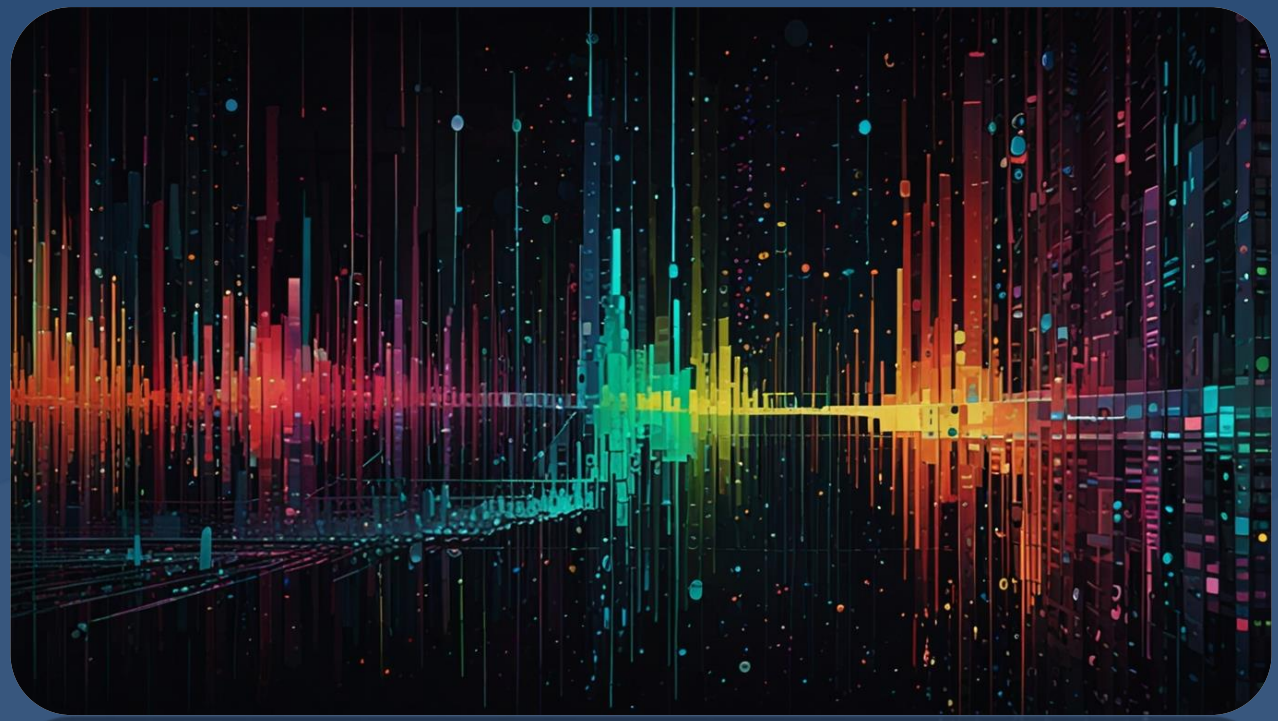


Span Slicing vs. Substrings

Span Slicing vs. Substrings

- Overview of ‘Span<T>’ and ‘Memory<T>’
- Advantages of Span Slicing

- Example:
  Replacing Substrings
  in Blazor



‘Span<T>’ and ‘Memory<T>’



- manage contiguous regions of arbitrary memory
- **Span<T>** operates on stack-allocated memory
- **Memory<T>** is the heap-allocated counterpart to **Span<T>**

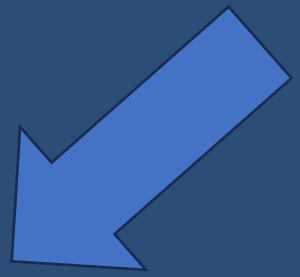
Advantages of Span Slicing



- Performance Gains
- Reduced Memory Allocations
- Improved Safety
- Flexibility and Versatility

Example: CSV parsing


```
var csvLine = "John,Doe,john.doe@example.com";
var firstName = csvLine.Substring(0, csvLine.IndexOf(',')); // Extracts "John"
var lastName = csvLine.Substring(csvLine.IndexOf(',') + 1, csvLine.LastIndexOf(',') - csvLine.IndexOf(',') - 1); // Extracts "Doe"
var email = csvLine.Substring(csvLine.LastIndexOf(',') + 1); // Extracts "john.doe@example.com"
```



```
var csvLine = "John,Doe,john.doe@example.com";
var span = csvLine.AsSpan();

// Find the indexes of the commas
var firstComma = span.IndexOf(',');
var lastComma = span.LastIndexOf(',');

// Slice the span to get each field without allocating new strings
var firstName = span.Slice(0, firstComma);
var lastName = span.Slice(firstComma + 1, lastComma - firstComma - 1);
var email = span.Slice(lastComma + 1);

// Convert to string if needed
var firstNameStr = firstName.ToString();
var lastNameStr = lastName.ToString();
var emailStr = email.ToString();
```



Array Pooling

Array Pooling



- Introduction to Array Pooling
- Benefits for Memory Management
- Implementation in Blazor

Introduction to Array Pooling

- Reuse arrays rather than allocating and deallocating
- Arrays are taken from and returned to a pool
- Increases processing and memory performance



Implementation Example

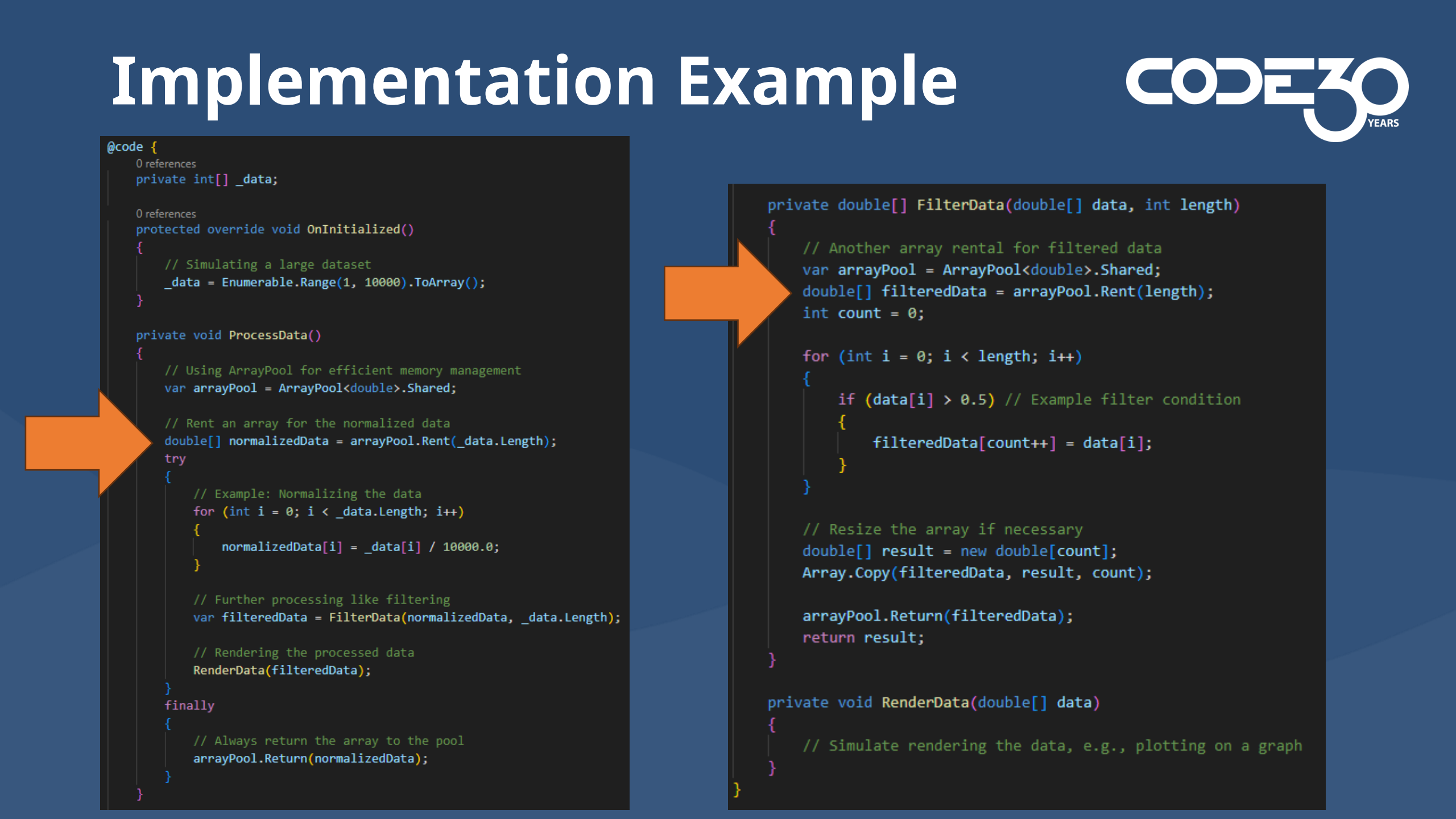


Scenario:

- Blazor component processes large dataset (e.g. graph, grid)
- Need for sorting, filtering, transforming arrays of data
- Without array pooling: operations would create many arrays

- The component needs to filter, normalize, and then render this data multiple times as the user interacts with it.

Implementation Example



```razor
@code {
    0 references
    private int[] _data;

    0 references
    protected override void OnInitialized()
    {
        // Simulating a large dataset
        _data = Enumerable.Range(1, 10000).ToArray();
    }

    private void ProcessData()
    {
        // Using ArrayPool for efficient memory management
        var arrayPool = ArrayPool<double>.Shared;

        // Rent an array for the normalized data
        double[] normalizedData = arrayPool.Rent(_data.Length);
        try
        {
            // Example: Normalizing the data
            for (int i = 0; i < _data.Length; i++)
            {
                normalizedData[i] = _data[i] / 10000.0;
            }

            // Further processing like filtering
            var filteredData = FilterData(normalizedData, _data.Length);

            // Rendering the processed data
            RenderData(filteredData);
        }
        finally
        {
            // Always return the array to the pool
            arrayPool.Return(normalizedData);
        }
    }
```

```razor
    private double[] FilterData(double[] data, int length)
    {
        // Another array rental for filtered data
        var arrayPool = ArrayPool<double>.Shared;
        double[] filteredData = arrayPool.Rent(length);
        int count = 0;

        for (int i = 0; i < length; i++)
        {
            if (data[i] > 0.5) // Example filter condition
            {
                filteredData[count++] = data[i];
            }
        }

        // Resize the array if necessary
        double[] result = new double[count];
        Array.Copy(filteredData, result, count);

        arrayPool.Return(filteredData);
        return result;
    }

    private void RenderData(double[] data)
    {
        // Simulate rendering the data, e.g., plotting on a graph
    }
}
```



Zero Copy Techniques

Zero Copy Techniques

- Concept of Zero Copy
- Use Cases and Benefits
- Practical Blazor Example



Concept of Zero Copy

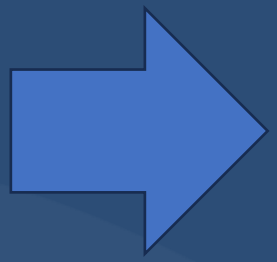


- Transfer data from one part of a system to another without being copied or temporary buffers
- Use pointers or references to the data instead

Use Cases and Benefits



- Network Programming
- Memory Mapped Files
- Blazor Components



- Performance Gains
- Efficiency
- Lower GC Pressure

Practical Example



- Use Span<T> and Memory<T> to work with slices of arrays or buffers

```
public void ProcessData(ReadOnlySpan<byte> data)
{
    // Process the data directly without copying
    foreach (var item in data)
    {
        // Perform some operation
    }
}
```

Another Practical Example

- Memory Mapping:

  large log or data file processing/analysis

```
using System;
using System.IO.MemoryMappedFiles;

1 reference
class Program
{
    0 references
    static void Main()
    {
        string filePath = "l      le.dat"; // Path to your large file

        // Create a memory-mapped file from an existing file
        using (var mmf = MemoryMappedFile.C      FromFile(filePath, FileMode.Open, "MyMappedFile"))
        {
            // Create a view accessor to access portions of the file
            using (var accessor = mmf.CreateViewAccessor(0, 0, MemoryMappedFileAccess.Read))
            {
                // Example: Read and process the first 10 bytes
                byte[] buffer = new byte[10];
                accessor.ReadArray(0, buffer, 0, buffer.Length);

                // Process the data (e.g., print bytes)
                Console.WriteLine("First 10 bytes of the file:");
                foreach (var b in buffer)
                {
                    Console.Write(b + " ");
                }
            }
        }
    }
}
```



Structs of Arrays

Structs of Arrays

- Understanding Structs of Arrays vs. Arrays of Structs
- Performance Implications
- Blazor Implementation Example

Arrays of Structs vs Structs of Arrays




```
struct Person
{
  int Age;
  double Height;
}
Person[] people;
```

Continuous blocks of memory for each struct, with fields interleaved

```
int[] ages; double[] heights;
```

Separate, contiguous arrays for each field

Performance Implications

- Cache Efficiency
- Vectorization
- GC Pressure



Example: Turn Arrays of Structs into Structs of Arrays




```
struct Entry { int ID; double Value; }
Entry[] entries = new Entry[1000];

// Access pattern
foreach(var entry in entries) { RenderEntry(entry); }
```



```
int[] ids = new int[1000];
double[] values = new double[1000];

// Access pattern
for(int i = 0; i < 1000; i++) { RenderEntry(ids[i], values[i]); }
```



Stackalloc

Stackalloc

- Introduction to Stack Allocation
- Performance Benefits and Risks
- Example: Using ‘stackalloc’ in Blazor



Introduction to Stack Allocation

- Allocate memory on the stack instead of the heap
- Allocate a fixed size small block of memory during compile time
- Memory is automatically freed when the method returns
- Ideal for scenarios requiring high-performance, low-allocation overhead, and small data structures.

Performance Benefits and Risks

- Speed
- Reduced GC Pressure
- No Fragmentation



- Limited Size
- Stack Overflow
- Unsafe Context

Using ‘stackalloc’ in Blazor




```
@code {
    void ProcessData()
    {
        int length = 100;
        Span<int> numbers = stackalloc int[length];

        for (int i = 0; i < length; i++)
        {
            numbers[i] = i * 2; // Simple operation for demo purposes
        }

        // Use numbers within this method
    }
}
```

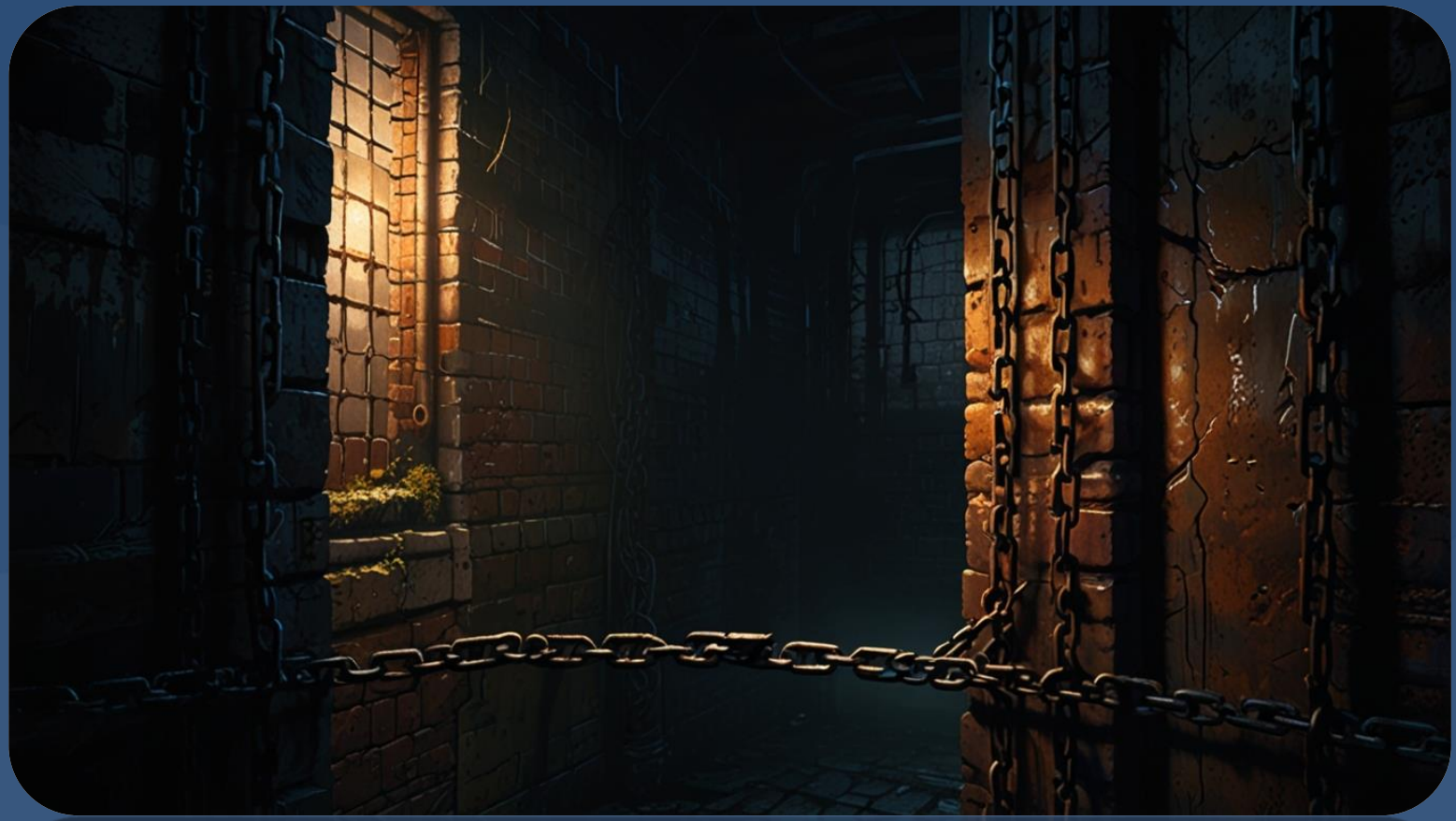


LINQ Avoidance

LINQ Avoidance

- LINQ: PROs and CONs
- Alternatives to LINQ in Performance Sensitive Code

- Example in a Blazor Context



LINQ: Pros and Cons

- Pros:
  - Readability
  - Expressiveness
  - Flexibility
- Cons:
  - Performance Overhead
  - Memory Usage
  - Debugging Complexity

Alternatives to LINQ in Performance-Sensitive Code



- Manual Loops
- Span<T> and Memory<T>
- Structs and Value Types
- Avoiding Deferred Execution

Example in a Blazor Context




```
var recentActivities = allActivities
    .Where(activity => activity.Date >= DateTime.Now.AddDays(-30))
    .GroupBy(activity => activity.UserId)
    .Select(group => new UserActivityGroup
    {
        UserId = group.Key,
        Activities = group.ToList()
    })
    .ToList();
```

Example in a Blazor Context



- **Reduced Iterations:**
Data is iterated over only twice

- **Greater Control:** More control over the processing flow, allowing for potential optimizations (e.g., early exits).

```
var recentActivities = new List<UserActivityGroup>();
var activityGroups = new Dictionary<int, List<Activity>>();

foreach (var activity in allActivities)
{
    if (activity.Date >= DateTime.Now.AddDays(-30))
    {
        if (!activityGroups.ContainsKey(activity.UserId))
        {
            activityGroups[activity.UserId] = new List<Activity>();
        }
        activityGroups[activity.UserId].Add(activity);
    }
}

foreach (var group in activityGroups)
{
    recentActivities.Add(new UserActivityGroup
    {
        UserId = group.Key,
        Activities = group.Value
    });
}
```



Garbage Collection Optimization

GC Collection Optimization



- Overview of .NET GC Mechanisms
- Techniques for Reducing GC Pressure
- Practical Tips for GC Optimization in Blazor

Overview of .NET GC Mechanisms

- Generational Garbage Collection
- Automatic Memory Management
- GC Modes
  (GC.TryStartNoGCRegion)



Techniques for Reducing GC Pressure

- Avoid Unnecessary Object Creation
- Use Structs Instead of Classes
- Array Pooling
- Span<T> & Memory<T>

- Minimize Component Re-renders
- Use IAsyncDisposable
- Avoid Boxing
- Profile and Tune



CODE Staffing

Disrupting the world of staffing!

Giving our customers the ability to have staff on par with Silicon Valley companies ...

... and our employees a work environment in a bleeding-edge tech company with the **industry leading benefits!**

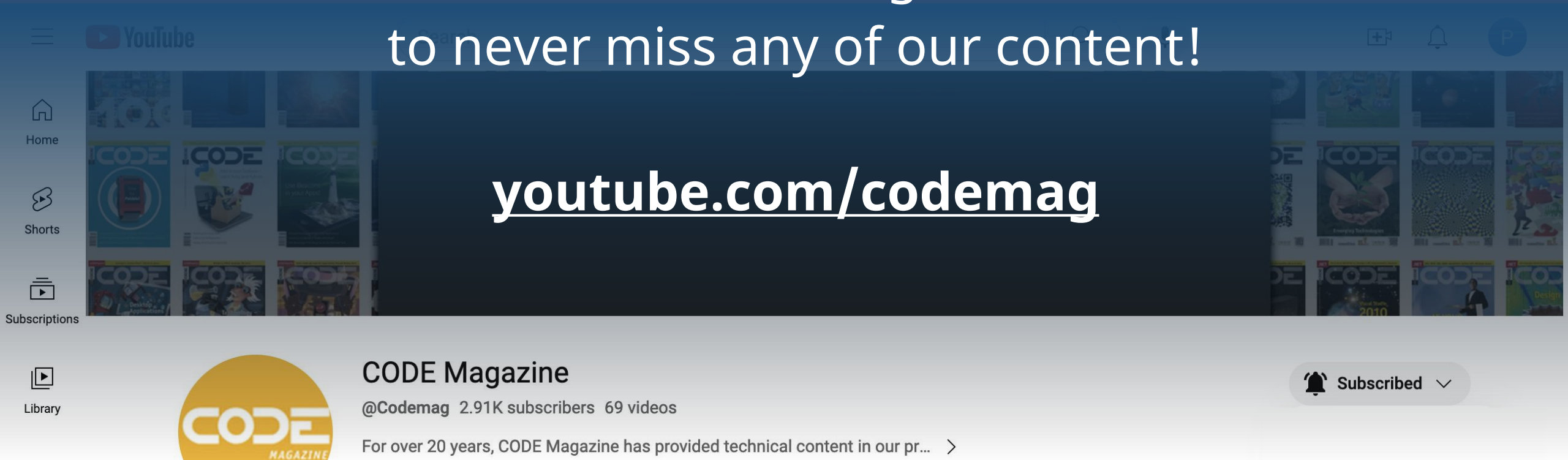
codestaffing.com



YouTube

Subscribe and “ring that bell”
to never miss any of our content!

youtube.com/codemag



Survey!



Subscribe!



Q&A

Contact us with questions!

CODE Contact

codemag.com
info@codemag.com
facebook.com/codemag
twitter.com/codemagazine

Presenter Contact

otto@codemag.com



@professormuscle