Tutorial: Reduce memory allocations with `ref` safety

Article • 04/07/2023

Often, performance tuning for a .NET application involves two techniques. First, reduce the number and size of heap allocations. Second, reduce how often data is copied. Visual Studio provides great tools that help analyze how your application is using memory. Once you've determined where your app makes unnecessary allocations, you make changes to minimize those allocations. You convert `class` types to `struct` types. You use `ref` safety features to preserve semantics and minimize extra copying.

Use Visual Studio 17.5 for the best experience with this tutorial. The .NET object allocation tool used to analyze memory usage is part of Visual Studio. You can use Visual Studio Code and the command line to run the application and make all the changes. However, you won't be able to see the analysis results of your changes.

The application you'll use is a simulation of an IoT application that monitors several sensors to determine if an intruder has entered a secret gallery with valuables. The IoT sensors are constantly sending data that measures the mix of Oxygen (O2) and Carbon Dioxide (CO2) in the air. They also report the temperature and relative humidity. Each of these values is fluctuating slightly all the time. However, when a person enters the room, the change a bit more, and always in the same direction: Oxygen decreases, Carbon Dioxide increases, temperature increases, as does relative humidity. When the sensors combine to show increases, the intruder alarm is triggered.

In this tutorial, you'll run the application, take measurements on memory allocations, then improve the performance by reducing the number of allocations. The source code is available in the samples browser.

Explore the starter application

Download the application and run the starter sample. The starter application works correctly, but because it allocates many small objects with each measurement cycle, its performance slowly degrades as it runs over time.

Console

```
Press <return> to start simulation

Debounced measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906
Average measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906

Debounced measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
Average measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
```

Many rows removed.

Console

```
Debounced measurements:
    Temp:      67.597
    Humidity:  46.543%
    Oxygen:    19.021%
    CO2 (ppm): 429.149
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High

Debounced measurements:
    Temp:      67.602
    Humidity:  46.835%
    Oxygen:    19.003%
    CO2 (ppm): 429.393
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High
```

You can explore the code to learn how the application works. The main program runs the simulation. After you press `<Enter>`, it creates a room, and gathers some initial baseline data:

C#

```csharp
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;

room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        Console.WriteLine();
        counter++;
        return counter < 20000;
    });
```

Once that baseline data has been established, it runs the simulation on the room, where a random number generator determines if an intruder has entered the room:

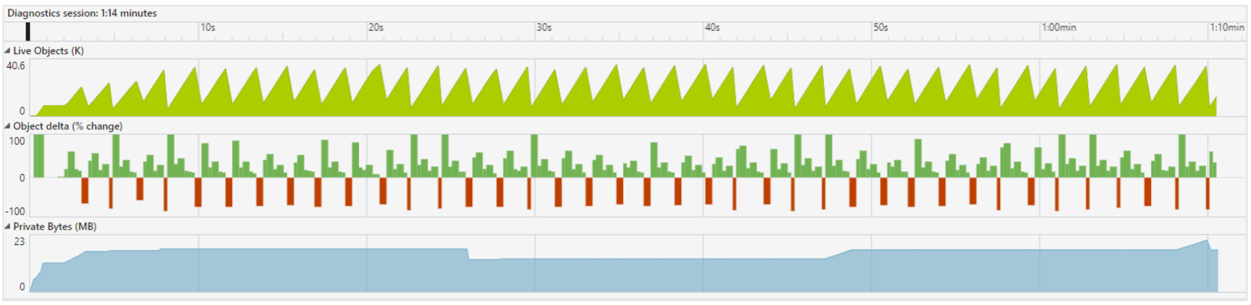
C#

```csharp
counter = 0;
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        room.Intruders += (room.Intruders, r.Next(5)) switch
        {
            ( > 0, 0) => -1,
            ( < 3, 1) => 1,
            _ => 0
        };

        Console.WriteLine($"Current intruders: {room.Intruders}");
        Console.WriteLine($"Calculated intruder risk:
{room.RiskStatus}");
        Console.WriteLine();
        counter++;
        return counter < 200000;
    });
```

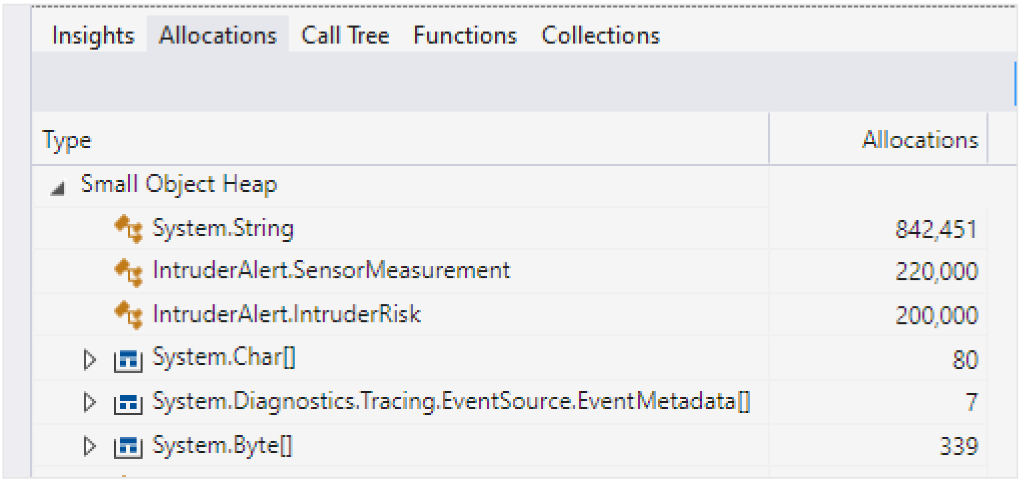
Other types contain the measurements, a debounced measurement that is the average of the last 50 measurements, and the average of all measurements taken.

Next, run the application using the .NET object allocation tool. Make sure you're using the `Release` build, not the `Debug` build. On the *Debug* menu, open the *Performance profiler*. Check the *.NET Object Allocation Tracking* option, but nothing else. Run your application to completion. The profiler measures object allocations and reports on allocations and garbage collection cycles. You should see a graph similar to the following image:



The previous graph shows that working to minimize allocations will provide performance benefits. You see a sawtooth pattern in the live objects graph. That tells you that numerous objects are created that quickly become garbage. They're later collected, as shown in the object delta graph. The downward red bars indicate a garbage collection cycle.

Next, look at the *Allocations* tab below the graphs. This table shows what types are allocated the most:



Type	Allocations
Small Object Heap	
System.String	842,451
IntruderAlert.SensorMeasurement	220,000
IntruderAlert.IntruderRisk	200,000
System.Char[]	80
System.Diagnostics.Tracing.EventSource.EventMetadata[]	7
System.Byte[]	339

The System.String type accounts for the most allocations. The most important task should be to minimize the frequency of string allocations. This application prints numerous formatted output to the console constantly. For this simulation, we want to keep messages, so we'll concentrate on the next two rows: the `SensorMeasurement` type, and the `IntruderRisk` type.

Double-click on the `SensorMeasurement` line. You can see that all the allocations take place in the `static` method `SensorMeasurement.TakeMeasurement`. You can see the method in the following snippet:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int in-
truders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5
* generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Every measurement allocates a new `SensorMeasurement` object, which is a `class` type. Every `SensorMeasurement` created causes a heap allocation.

Change classes to structs

The following code shows the initial declaration of `SensorMeasurement`:

C#

```
public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int
intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) +
(0.5 * generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20
* generator.NextDouble() - 0.10),
            Room = room,
            TimeRecorded = DateTime.Now
        };
    }

    private const double CO2Concentration = 409.8; // increases with
people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
    public required double Humidity { get; init; }
    public required string Room { get; init; }
    public required DateTime TimeRecorded { get; init; }

    public override string ToString() => $"""
            Room: {Room} at {TimeRecorded}:
                Temp:      {Temperature:F3}
                Humidity:  {Humidity:P3}
                Oxygen:    {O2:P3}
                CO2 (ppm): {CO2:F3}
            """;
}
```

The type was originally created as a `class` because it contains numerous `double` measurements. It's larger than you'd want to copy in hot paths. However, that decision meant a large number of allocations. Change the type from a `class` to a `struct`.

Changing from a `class` to `struct` introduces a few compiler errors because the original code used `null` reference checks in a few spots. The first is in the `DebounceMeasurement` class, in the `AddMeasurement` method:

C#

```csharp
public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize
: totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? de-
bounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
}
```

The `DebounceMeasurement` type contains an array of 50 measurements. The readings for a sensor are reported as the average of the last 50 measurements. That reduces the noise in the readings. Before a full 50 readings have been taken, these values are `null`. The code checks for `null` reference to report the correct average on system startup. After changing the `SensorMeasurement` type to a struct, you must use a different test. The `SensorMeasurement` type includes a `string` for the room identifier, so you can use that test instead:

C#

```csharp
if (recentMeasurements[i].Room is not null)
```

The other three compiler errors are all in the method that repeatedly takes measurements in a room:

C#

```csharp
public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
```

In the starter method, the local variable for the `SensorMeasurement` is a *nullable reference*:

C#

```
SensorMeasurement? measure = default;
```

Now that the `SensorMeasurement` is a `struct` instead of a `class`, the nullable is a *nullable value type*. You can change the declaration to a value type to fix the remaining compiler errors:

C#

```
SensorMeasurement measure = default;
```

Now that the compiler errors have been addressed, you should examine the code to ensure the semantics haven't changed. Because `struct` types are passed by value, modifications made to method parameters aren't visible after the method returns.

ⓘ **Important**

Changing a type from a `class` to a `struct` can change the semantics of your program. When a `class` type is passed to a method, any mutations made in the method are made to the argument. When a `struct` type is passed to a method, and mutations made in the method are made to *a copy* of the argument. That means any method that modifies its arguments by design should be updated to use the `ref` modifier on any argument type you've changed from a `class` to a `struct`.

The `SensorMeasurement` type doesn't include any methods that change state, so that's not a concern in this sample. You can prove that by adding the `readonly` modifier to the `SensorMeasurement` struct:

C#

```
public readonly struct SensorMeasurement
```

The compiler enforces the `readonly` nature of the `SensorMeasurement` struct. If your inspection of the code missed some method that modified state, the compiler would tell you. Your app still builds without errors, so this type is `readonly`. Adding the `readonly` modifier when you change a type from a `class` to a `struct` can help you find members that modify the state of the `struct`.

Avoid making copies

You've removed a large number of unnecessary allocations from your app. The `SensorMeasurement` type doesn't appear in the table anywhere.

Now, it's doing extra working copying the `SensorMeasurement` structure every time it's used as a parameter or a return value. The `SensorMeasurement` struct contains four doubles, a DateTime and a `string`. That structure is measurably larger than a reference. Let's add the `ref` or `in` modifiers to places where the `SensorMeasurement` type is used.

The next step is to find methods that return a measurement, or take a measurement as an argument, and use references where possible. Start in the `SensorMeasurement` struct. The static `TakeMeasurement` method creates and returns a new `SensorMeasurement`:

C#

```csharp
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 * generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 * generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 * generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 * generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

We'll leave this one as is, returning by value. If you tried to return by `ref`, you'd get a compiler error. You can't return a `ref` to a new structure locally created in the method. The design of the immutable struct means you can only set the values of the measurement at construction. This method must create a new measurement struct.

Let's look again at `DebounceMeasurement.AddMeasurement`. You should add the `in` modifier to the `measurement` parameter:

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize : 
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize 
: totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? de-
bounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? 
debounceSize : totalMeasurements);
}
```

That saves one copy operation. The `in` parameter is a reference to the copy already created by the caller. You can also save a copy with the `TakeMeasurement` method in the `Room` type. This method illustrates how the compiler provides safety when you pass arguments by `ref`. The initial `TakeMeasurement` method in the `Room` type takes an argument of `Func<SensorMeasurement, bool>`. If you try to add the `in` or `ref` modifier to that declaration, the compiler reports an error. You can't pass a `ref` argument to a lambda expression. The compiler can't guarantee that the called expression doesn't copy the reference. If the lambda expression *captures* the reference, the reference could have a lifetime longer than the value it refers to. Accessing it outside its *ref safe to escape scope* would result in memory corruption. The `ref` safety rules don't allow it. You can learn more in the overview of ref safety features.

Preserve semantics

The final sets of changes won't have a major impact on this application's performance because the types aren't created in hot paths. These changes illustrate some of the other techniques you'd use in your performance tuning. Let's take a look at the initial `Room` class:

C#

```csharp
public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }

    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 /
4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 /
4.0;
            var TempVariance = (Debounce.Temperature -
Average.Temperature) > 0.05 / 4.0;
            var HumidityVariance = (Debounce.Humidity -
Average.Humidity) > 0.20 / 4;
            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }
            return risk;
        }
    }

    public int Intruders { get; set; }


    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name,
Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}
```

This type contains several properties. Some are `class` types. Creating a `Room` object involves multiple allocations. One for the `Room` itself, and one for each of the members of a `class` type that it contains. You can convert two of these properties from `class` types to `struct` types: the `DebounceMeasurement` and `AverageMeasurement` types. Let's work through that transformation with both types.

Change the `DebounceMeasurement` type from a `class` to `struct`. That introduces a compiler error `CS8983: A 'struct' with field initializers must include an explicitly declared constructor`. You can fix this by adding an empty parameterless constructor:

```C#
public DebounceMeasurement() { }
```

You can learn more about this requirement in the language reference article on structs.

The Object.ToString() override doesn't modify any of the values of the struct. You can add the `readonly` modifier to that method declaration. The `DebounceMeasurement` type is *mutable*, so you'll need to take care that modifications don't affect copies that are discarded. The `AddMeasurement` method does modify the state of the object. It's called from the `Room` class, in the `TakeMeasurements` method. You want those changes to persist after calling the method. You can change the `Room.Debounce` property to return a *reference* to a single instance of the `DebounceMeasurement` type:

```C#
private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref
debounce; } }
```

There are a few changes in the previous example. First, the *property* is a readonly property that returns a readonly reference to the instance owned by this room. It's now backed by a declared field that's initialized when the `Room` object is instantiated. After making these changes, you'll update the implementation of `AddMeasurement` method. It uses the private backing field, `debounce`, not the readonly property `Debounce`. That way, the changes take place on the single instance created during initialization.

The same technique works with the `Average` property. First, you modify the `AverageMeasurement` type from a `class` to a `struct`, and add the `readonly` modifier on the `ToString` method:

C#

```csharp
namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature / totalMea-
surements;
    public readonly double Humidity => sumHumidity / totalMeasure-
ments;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:       {Temperature:F3}
            Humidity:   {Humidity:P3}
            Oxygen:     {O2:P3}
            CO2 (ppm):  {CO2:F3}
        """;
}
```

Then, you modify the `Room` class following the same technique you used for the `Debounce` property. The `Average` property returns a `readonly ref` to the private field for the average measurement. The `AddMeasurement` method modifies the internal fields.

C#

```csharp
private AverageMeasurement average = new();
public  ref readonly AverageMeasurement Average { get { return ref
average; } }
```

Avoid boxing

There's one final change to improve performance. The main program is printing stats for the room, including the risk assessment:

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

The call to the generated `ToString` boxes the enum value. You can avoid that by writing an override in the `Room` class that formats the string based on the value of estimated risk:

C#

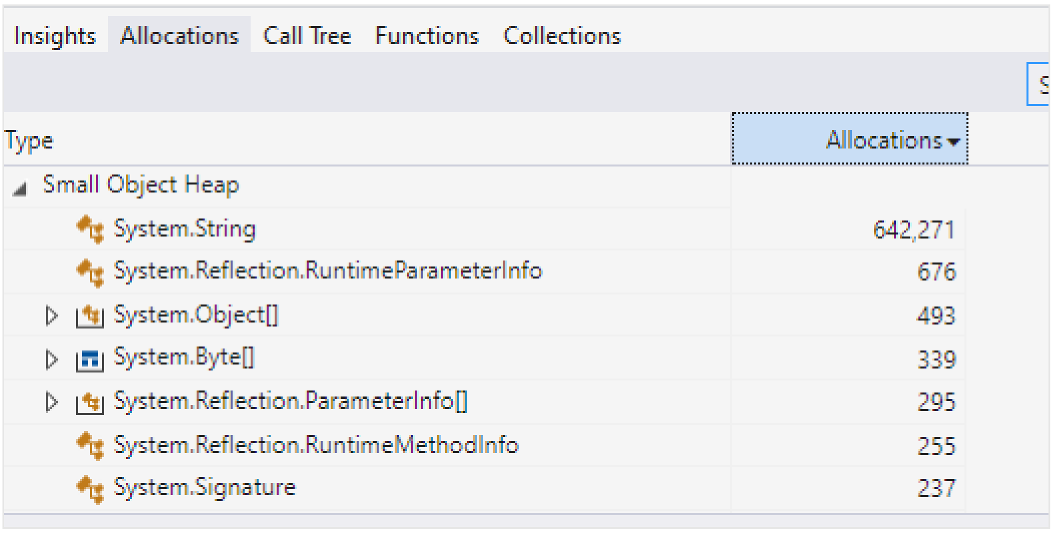
```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

Then, modify the code in the main program to call this new `ToString` method:

C#

```
Console.WriteLine(room.ToString());
```

Run the app using the profiler and look at the updated table for allocations.



Type	Allocations
Small Object Heap	
System.String	642,271
System.Reflection.RuntimeParameterInfo	676
System.Object[]	493
System.Byte[]	339
System.Reflection.ParameterInfo[]	295
System.Reflection.RuntimeMethodInfo	255
System.Signature	237

You've removed numerous allocations, and provided your app with a performance boost.

Using ref safety in your application

These techniques are low-level performance tuning. They can increase performance in your application when applied to hot paths, and when you've measured the impact before and after the changes. In most cases, the cycle you'll follow is:

- *Measure allocations*: Determine what types are being allocated the most, and when you can reduce the heap allocations.
- *Convert class to struct*: Many times, types can be converted from a `class` to a `struct`. Your app uses stack space instead of making heap allocations.
- *Preserve semantics*: Converting a `class` to a `struct` can impact the semantics for parameters and return values. Any method that modifies its parameters should now mark those parameters with the `ref` modifier. That ensures the modifications are made to the correct object. Similarly, if a property or method return value should be modified by the caller, that return should be marked with the `ref` modifier.
- *Avoid copies*: When you pass a large struct as a parameter, you can mark the parameter with the `in` modifier. You can pass a reference in fewer bytes, and ensure that the method doesn't modify the original value. You can also return values by `readonly ref` to return a reference that can't be modified.

Using these techniques you can improve performance in hot paths of your code.