blazar

MANUEL MARQUEZ

manuelxmarquez@gmail.com
manuelmarquez.ca

Contents

Purpose

The primary purpose of this document is to highlight my experience with compilers and LLVM. This work, created entirely by myself, demonstrates the ability to design a language pipeline, implement a compiler, and integrate with LLVM for high-performance code generation.

Motivation

The project began as an experiment to analyze performance differences between C# and native code in my game engine, CyberEngine. Initially, Roslyn was used to automatically convert portions of the engine to C++, leveraging experience gained from building a VB.NET to C# code converter. While the approach worked, compiling to C++ introduced significant overhead, making it clear that producing LLVM IR and machine code directly would streamline the process.

Overview

Blazar is a programming language implemented in C#. Its front end is based on a fork of Roslyn, extended with additional keywords and custom language semantics. The compiler middle end takes Roslyn's bound syntax nodes and lowers them to a custom stack-based intermediate language (BIL), which is then translated into LLVM IR using the LLVMSharp C# wrapper.

The language and compiler support structs, heap-allocated objects, arrays, object fields, instance methods, virtual functions with method overriding, conditional branching, loops, native C interoperability, console I/O, file streams, exception handling with try/catch/finally, and additional core language features.

The project system is built on MSBuild and supports project references. A custom Visual Studio extension provides integrated build, debugging, and editor support, including syntax highlighting, code formatting, code completion, and syntactic analysis. The compiler also supports Linux, enabling cross-platform development and execution.

Language

Blazar source code closely resembles C#, as its front end is based on a fork of the Roslyn compiler. This means it inherits many of C#'s familiar language constructs, such as classes, interfaces, inheritance, and virtual methods. The compiler pipeline begins by parsing the source code into a syntax tree, which is then processed by the binder to perform semantic analysis. After resolving symbols and types, a graph of bound nodes is produced, which serve as the input for Blazar's middle end compilation stages, where the code is lowered into an intermediate representation.

SOURCE CODE

The following Blazar source code calculates the sum of squares from 1 to 9 and outputs the intermediate and final results to the console.

```
public class Program
{
    public static unsafe int Main()
    {
        var sum = 0;

        for (var i = 1; i < 10; i++)
        {
            sum = sum + i * i;
            Console.Write("i = ");
            Console.Write(i);
            Console.Write("  sum = ");
            Console.WriteLine(sum);
        }

        Console.Write("Final sum of squares: ");
        Console.WriteLine(sum);

        return 0;
    }
}
```

Figure 1 Source code for calculating the sum of squares.

INTERMEDIATE LANGUAGE

Blazar Intermediate Language (BIL) is a stack-based intermediate representation used by the compiler for optimization and code generation. Each function is divided into blocks of linear sequences of instructions that must begin at the first instruction in the block and end with a control flow terminator such as a branch or return. This structure simplifies optimization and validation passes by providing well-defined control flow. The choice to use an intermediate language also enables future support for additional front ends while reusing the same optimization and code generation infrastructure.

In the following example, the `Main` function from the sum of squares program is shown after being lowered into Blazar's intermediate representation. The variable `LoweringTemp0` is a temporary variable introduced during the lowering process to hold an intermediate computation. However, since it is written to at instruction `0019` but never subsequently read, it represents dead storage that could be safely removed by a dead variable elimination optimization pass.

```
public static int [Blazar.Test]Program.Main()
{
   int sum
   int i
   int LoweringTemp0

   0000: LoadConstantInt32 0
   0001: StoreVariable sum
   0002: LoadConstantInt32 1
   0003: StoreVariable i
   0004: BranchAlways 0005

   0005: LoadVariable i
   0006: LoadConstantInt32 10
   0007: CompareLessThan
   0008: BranchConditional 0009, 001F

   0009: LoadVariable sum
   000A: LoadVariable i
   000B: LoadVariable i
   000C: Multiply
   000D: Add
   000E: StoreVariable sum
   000F: LoadConstantString "i = "
   0010: Call void [Blazar.System.Console]System.Console.Write(constr)
   0011: LoadVariable i
   0012: Call void [Blazar.System.Console]System.Console.Write(int)
   0013: LoadConstantString "  sum = "
   0014: Call void [Blazar.System.Console]System.Console.Write(constr)
   0015: LoadVariable sum
   0016: Call void [Blazar.System.Console]System.Console.WriteLine(int)
   0017: BranchAlways 0018

   0018: LoadVariable i
   0019: StoreVariable LoweringTemp0
   001A: LoadVariable i
   001B: LoadConstantInt32 1
   001C: Add
   001D: StoreVariable i
   001E: BranchAlways 0005

   001F: LoadConstantString "Final sum of squares: "
   0020: Call void [Blazar.System.Console]System.Console.Write(constr)
   0021: LoadVariable sum
   0022: Call void [Blazar.System.Console]System.Console.WriteLine(int)
   0023: LoadConstantInt32 0
   0024: ReturnValue
}
```
Figure 2 BIL for calculating the sum of squares.

LLVM

The back end uses LLVMSharp to translate BIL into LLVM IR, enabling Blazar to target multiple architectures. Blazar can also emit debug information alongside the generated IR, allowing inspection of variables during execution. Integration with LLVM further provides access to its advanced optimization pipeline and facilitates the generation of portable binaries across different platforms.

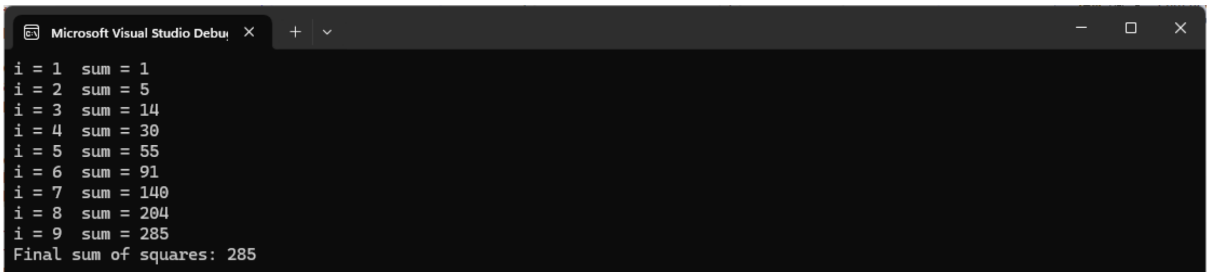
In the following example, the `Main` function from the sum of squares example is lowered into LLVM IR, then optimized and linked with core runtime assemblies to produce the final executable.

```
define i32 @"[Blazar.Test]Program.Main.2841951496"() {
entry:
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  %LoweringTemp0 = alloca i32, align 4
  store i32 0, ptr %sum, align 4
  store i32 1, ptr %i, align 4
  br label %b0

b0:                                               ; preds = %b2, %entry
  %0 = load i32, ptr %i, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %b1, label %b3

b1:                                               ; preds = %b0
  %1 = load i32, ptr %sum, align 4
  %2 = load i32, ptr %i, align 4
  %3 = load i32, ptr %i, align 4
  %mul = mul nsw i32 %2, %3
  %add = add nsw i32 %1, %mul
  store i32 %add, ptr %sum, align 4
  %4 = load %"[Blazar.Core]System.StringConstant", ptr @"[Blazar.Test].strc.2", align 1
  call void @"[Blazar.System.Console]System.Console.Write.2896451628"(%"[Blazar.Core]System.StringCon-
stant" %4)
  %5 = load i32, ptr %i, align 4
  call void @"[Blazar.System.Console]System.Console.Write.1741313647"(i32 %5)
  %6 = load %"[Blazar.Core]System.StringConstant", ptr @"[Blazar.Test].strc.4", align 1
  call void @"[Blazar.System.Console]System.Console.Write.2896451628"(%"[Blazar.Core]System.StringCon-
stant" %6)
  %7 = load i32, ptr %sum, align 4
  call void @"[Blazar.System.Console]System.Console.WriteLine.1741313647"(i32 %7)
  br label %b2

b2:                                               ; preds = %b1
  %8 = load i32, ptr %i, align 4
  store i32 %8, ptr %LoweringTemp0, align 4
  %9 = load i32, ptr %i, align 4
  %add1 = add nsw i32 %9, 1
  store i32 %add1, ptr %i, align 4
  br label %b0

b3:                                               ; preds = %b0
  %10 = load %"[Blazar.Core]System.StringConstant", ptr @"[Blazar.Test].strc.6", align 1
  call void @"[Blazar.System.Console]System.Console.Write.2896451628"(%"[Blazar.Core]System.StringCon-
stant" %10)
```
Figure 3 LLVM IR for calculating the sum of squares.

Linking

The final stage in Blazar uses the LLVM toolchain to link all project reference libraries and the compiled LLVM IR into a final library or executable.

The sum of squares example program displays the intermediate values of `i` and the running total sum at each iteration, then prints the final sum of squares to the terminal.



```
i = 1  sum = 1
i = 2  sum = 5
i = 3  sum = 14
i = 4  sum = 30
i = 5  sum = 55
i = 6  sum = 91
i = 7  sum = 140
i = 8  sum = 204
i = 9  sum = 285
Final sum of squares: 285
```

Figure 4 Console output after calculating the sum of squares.

Native Interoperability

Blazar supports integration with native C libraries by allowing managed functions to override their target symbol names using attributes. This enables direct calls to operating system and C runtime functions without additional wrappers or bindings.

For example, writing a character to a file is implemented using the standard C library function `fputc`, which is defined in Blazar with a `MethodName` attribute and overrides the default method name once lowered to LLVM.

```
[System.Runtime.MethodName("fputc")]
private static extern int WriteCharacter(int @char, nint stream);
```

Figure 5 Definition of fputc in Blazar.

```
declare i32 @fputc(i32, i64)
```

Figure 6 Lowering of fputc in LLVM.

During the linking stage, the function reference is resolved to the native implementation, enabling seamless interoperation with system-level libraries.

Objects

Blazar uses single inheritance, so each class can have at most one base class. Every object contains a pointer to its `TypeInfo` followed by the object's fields. Base class fields appear first, followed by derived class fields. The `TypeInfo` contains references to the virtual method table (VMT), ancestor tables, and optional interface descriptors, enabling runtime type checks and virtual dispatch.

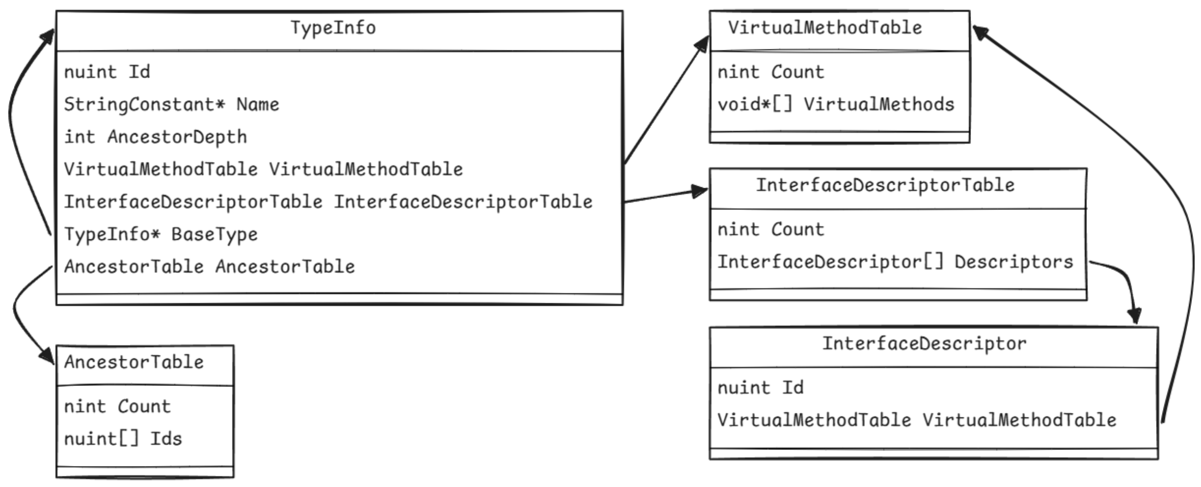


Figure 7 Class diagram for TypeInfo.

Consider the following example, which defines an abstract `Shape` base class with three derived types: `Circle`, `Square`, and `Sphere`. Each subclass overrides the virtual `Area` method to provide a specialized implementation based on its geometry. `Circle` and `Square` both implement the `IArea` interface, while `Sphere` additionally implements `IVolume` to provide volume calculations.

```
public interface IArea
{
    public double Area();
}

public interface IVolume
{
    public double Volume();
}

public abstract class Shape
{
    public int TypeId { get; }

    protected Shape(int typeId)
    {
        TypeId = typeId;
    }

    public virtual double Area() => 0.0;
}

public class Circle : Shape, IArea
{
    public double Radius { get; }

    public Circle(double radius) : base(1)
    {
        Radius = radius;
    }

    public override double Area() => Math.PI * Radius * Radius;
}

public class Square : Shape, IArea
{
    public double Size { get; }

    public Square(double size) : base(2)
    {
        Size = size;
    }

    public override double Area() => Size * Size;
}

public class Sphere : Shape, IArea, IVolume
{
    public double Radius { get; }

    public Sphere(double radius) : base(3)
    {
        Radius = radius;
    }

    public override double Area() => Math.PI * Radius * Radius;
    public double Volume() => (4.0 / 3.0) * Math.PI * Radius * Radius * Radius;
}
```

Figure 8 Source code for Shape, Circle, and Square.

The simplified `TypeInfo` definitions for `Shape`, `Circle`, `Square`, and `Sphere` describe how each type is represented in memory.

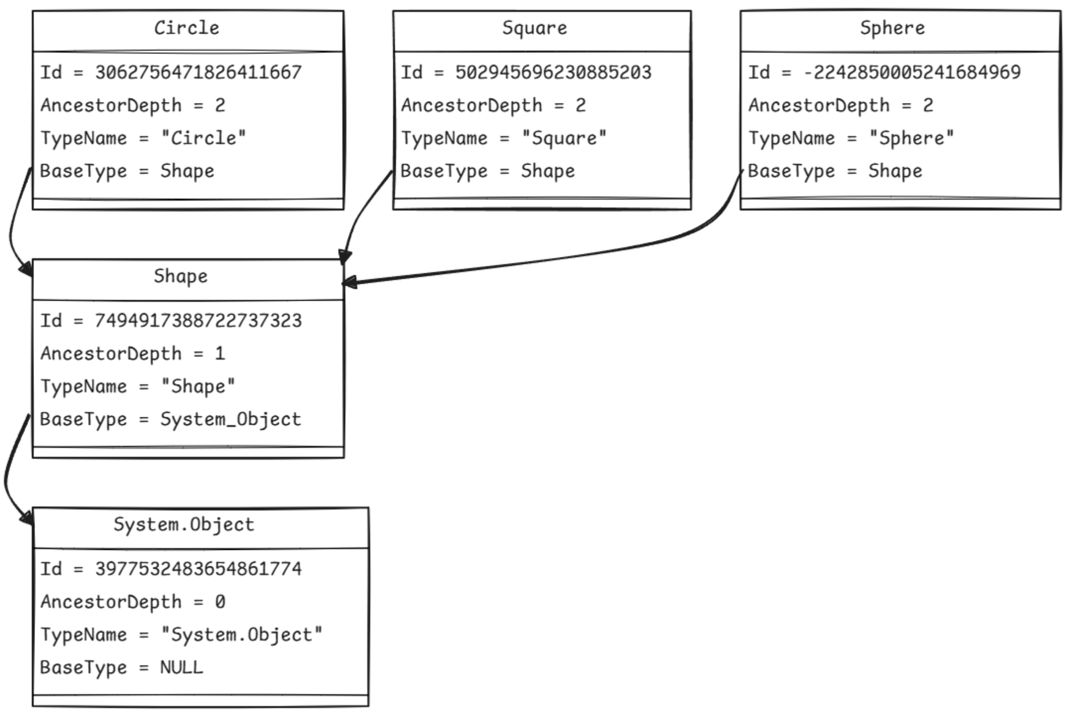


Figure 9 TypeInfo for Shape, Circle, and Square.

The main program creates a `Circle` and `Square`, then prints its information through a helper function that accepts a `Shape`. This involves an implicit upcast from the derived type to `Shape`, and the call to `Area()` is dispatched virtually at runtime based on the actual object type.

```
public class Program
{
    public static int Main()
    {
        var circle = new Circle(2);
        var square = new Square(2);
        var sphere = new Sphere(2);

        WriteType(circle);
        WriteArea(circle);

        WriteType(square);
        WriteArea(square);

        WriteType(sphere);
        WriteArea(sphere);
        WriteVolume(sphere);

        return 0;
    }

    private static void WriteType(Shape shape)
    {
        Console.Write("TypeId = ");
        Console.WriteLine(shape.TypeId);
    }

    private static void WriteArea(IArea area)
    {
        Console.Write("Area = ");
        Console.WriteLine(area.Area());
    }

    private static void WriteVolume(IVolume volume)
    {
        Console.Write("Volume = ");
        Console.WriteLine(volume.Volume());
    }
}
```

Figure 10 Source code creating shapes and printing its area.

When the program is executed, the following output is produced.

```
TypeId = 1
Area = 12.566370614359172
TypeId = 2
Area = 4
TypeId = 3
Area = 12.566370614359172
Volume = 33.510321638291124
```

Figure 11 Output for shape program.

The following diagram illustrates how `Circle`, `Square` and `Sphere` instances are laid out in memory. Each object begins with a pointer to its corresponding `TypeInfo`, followed by its instance fields.

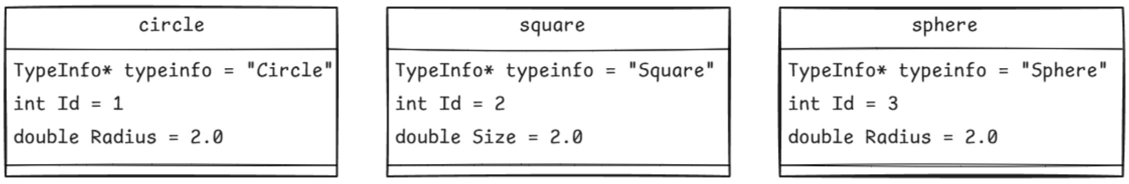


Figure 12 Memory layout of Circle and Square.

CASTING

Casting from a derived class to its base class incurs no runtime cost due to Blazar's single inheritance model. The validity of such a cast can be fully determined at compile time. Additionally, field offsets remain fixed because derived classes append their own fields after the base class's fields. This layout allows a base class function implementation to operate uniformly on all derived class instances.

Casting from a base class to a derived class, however, requires a runtime check to ensure the object is actually of the target type. This is accomplished by accessing the `TypeInfo` pointer, which is always the first field in every object, and verifying that the target type exists in the object's ancestor table. A simple comparison of type IDs is insufficient, as the object could be an instance of a further derived type.

Because of single inheritance, if the ancestor table is ordered from base to derived, the cast can be optimized by using the target type's ancestor depth to directly index into the table and verify the type ID with a single comparison.

The following diagram shows the `TypeInfo` and `AncestorTable` structures used to determine whether an object instance can be safely cast to a `Circle`.

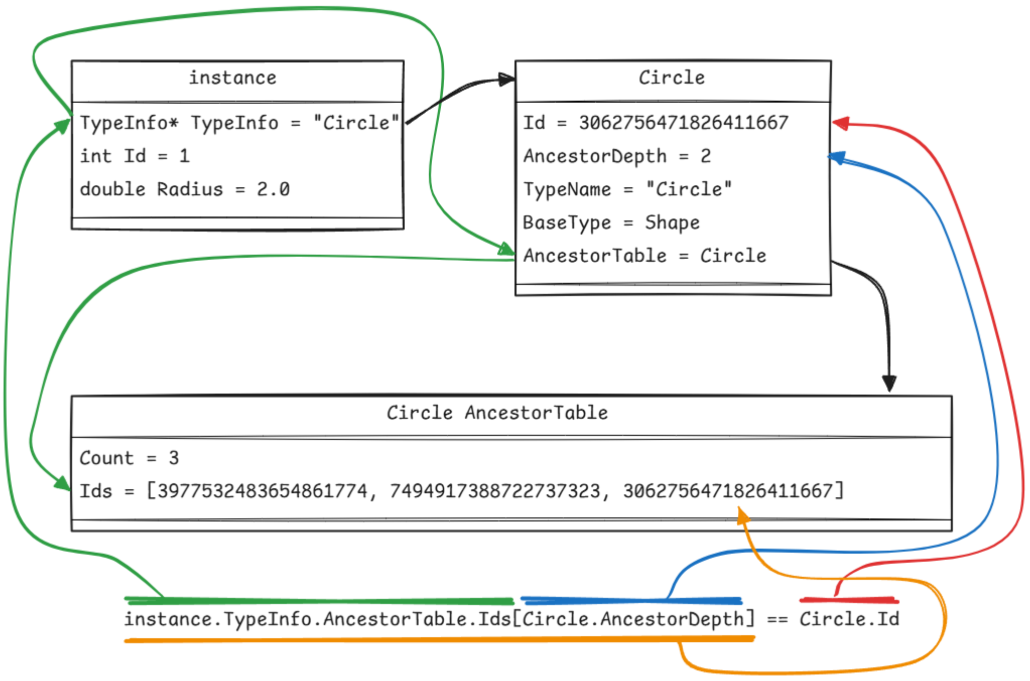


Figure 13 Type cast check.

The following code snippet demonstrates the implementation of the casting function, which accepts an object pointer and a pointer to the target `TypeInfo`.

```
internal static unsafe void* CastDynamic(void* instance, in TypeInfo castTypeInfo)
{
    if (instance == null)
        return null;

    var typeInfo = *(TypeInfo**)instance;

    if (castTypeInfo.AncestorDepth >= typeInfo->AncestorTable.Count ||
        typeInfo->AncestorTable.Ids[castTypeInfo.AncestorDepth] != castTypeInfo.Id)
        return null;

    return instance;
}
```

Figure 14 Source code to cast object instances.

VIRTUAL METHODS

Virtual methods are dispatched by first accessing the object's `TypeInfo` and then indexing into its virtual method table (VMT). Because Blazar uses single inheritance, the order of virtual methods is consistent across all derived types, allowing the table index to be determined at compile time. This enables the same lowered code to be used for all derived types when calling virtual methods.

The following diagram illustrates the relationship between an object's `TypeInfo` and its virtual method table.

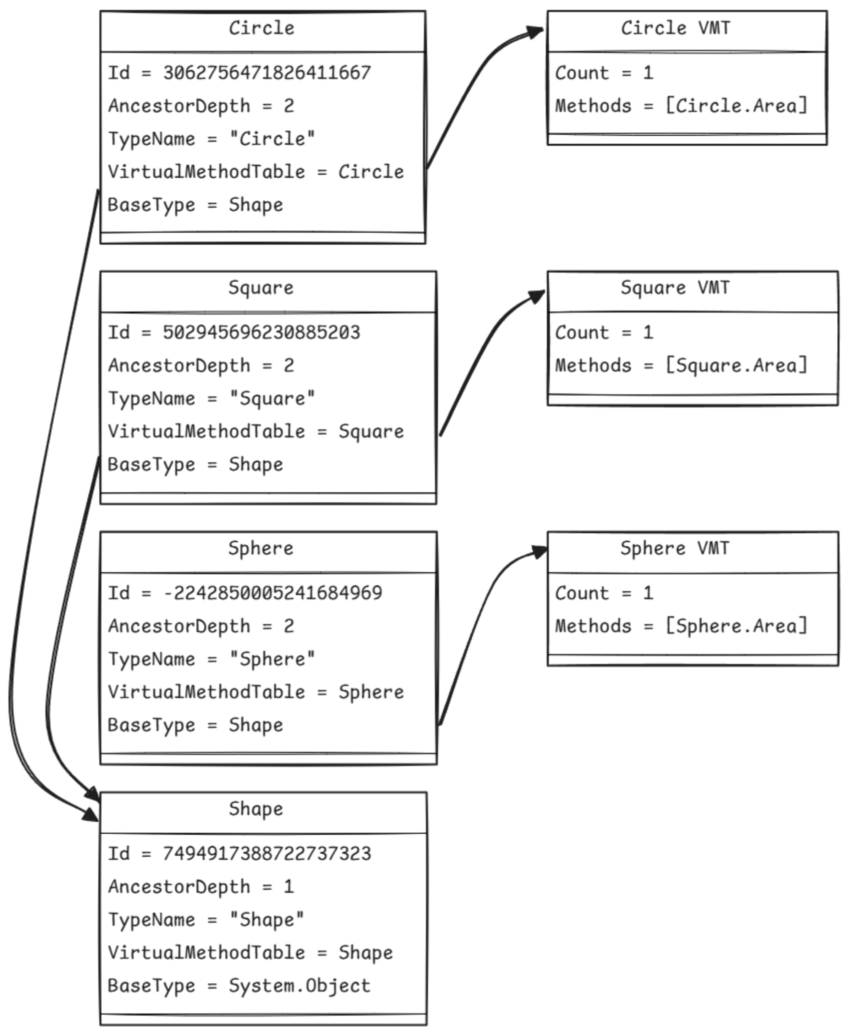
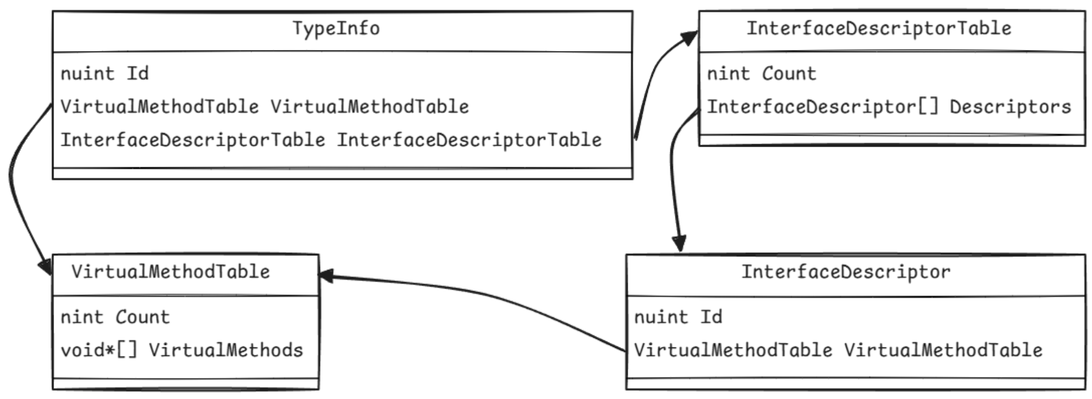


Figure 15 TypeInfo and virtual method table for Shape, Circle, and Square.

INTERFACES

Interfaces differ from classes because a single class can implement multiple interfaces. Each class contains an `InterfaceDescriptorTable`, which holds a list of `InterfaceDescriptor`s representing the interfaces it implements. Each `InterfaceDescriptor` stores the interface ID along with the corresponding virtual method table for that interface. This structure allows interface method calls to be correctly dispatched to the appropriate implementation for each object instance.


TypeInfo
nuint Id
VirtualMethodTable VirtualMethodTable
InterfaceDescriptorTable InterfaceDescriptorTable
InterfaceDescriptorTable
nint Count
InterfaceDescriptor[] Descriptors
VirtualMethodTable
nint Count
void*[] VirtualMethods
InterfaceDescriptor
nuint Id
VirtualMethodTable VirtualMethodTable
Figure 16 Class diagram for InterfaceDescriptorTable.

To avoid repeatedly retrieving the virtual method table for an interface at every call site, the lookup is performed once when the object is cast to an interface. At that point, an `InterfacePointer` is created, which contains both a pointer to the object instance and a pointer to the corresponding virtual method table. This allows subsequent interface method calls to dispatch directly through the `InterfacePointer` without additional type lookups.

```
internal unsafe struct InterfacePointer
{
    public void* Pointer;
    public void** VirtualMethods;
}
```

Figure 17 Source code for an interface pointer.

The following diagram illustrates the relationship between an object's `TypeInfo` and its associated `InterfaceDescriptors`, showing how each implemented interface links to its corresponding virtual method table.

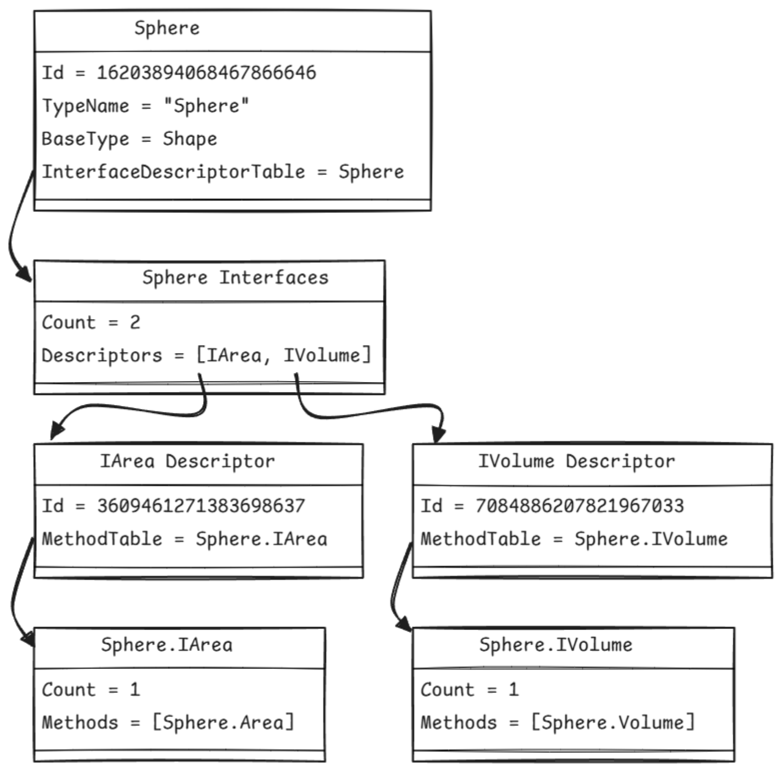


Figure 18 InterfaceDescriptorTable for Shape, Circle, and Square.

Casting an object to an interface requires iterating over the object's `InterfaceDescriptorTable` to locate the descriptor with the matching interface ID. Once found, an `InterfacePointer` can be constructed, containing a pointer to the object and a pointer to the interface's virtual method table.

```
internal static unsafe InterfacePointer CastInterface(void* instance, in TypeInfo castTypeInfo)
{
    if (instance != null)
    {
        var typeInfo = *(TypeInfo**)instance;

        var interfaceTable = typeInfo->InterfaceDescriptorTable;

        for (var i = 0; i < interfaceTable.Count; i++)
        {
            var descriptor = interfaceTable.Descriptors[i];

            if (descriptor.Id == castTypeInfo.Id)
                return new InterfacePointer
                {
                    Pointer = instance,
                    VirtualMethods = descriptor.VirtualMethodTable.VirtualMethods,
                };
        }
    }

    return default;
}
```

Figure 19 Source code to cast object instances to interface pointers.

VISUAL STUDIO

Supports a custom project type built on MSBuild with full editor tooling, including syntax highlighting, code formatting, code completion, code outlining, signature helper, go to definition, and code refactoring extensions.

SOLUTION EXPLORER

Custom project node in Solution Explorer with in-place project editing, allowing project files to be modified without unloading. Includes custom icons for Blazar source files.

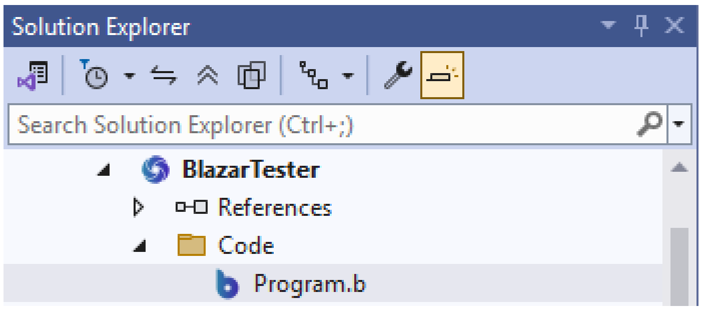


Figure 20 Blazar project in Solution Explorer.

MSBUILD

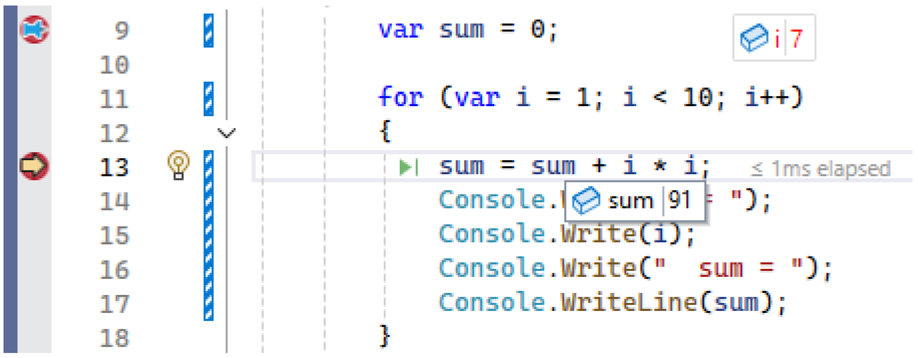
Project system built on MSBuild with support for `ProjectReference` and `PackageReference`, providing full compatibility with standard project file features used in MSBuild projects.

```xml
<Project Sdk="Mapromar.Blazar.Sdk">
  <PropertyGroup>
    <AssemblyName>BlazarTester</AssemblyName>
    <RootNamespace>BlazarTester</RootNamespace>
    <OutputType>Exe</OutputType>
    <TargetFramework>Blazar,Version=v1.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <Product>BlazarTester</Product>
    <AssemblyTitle>$(Product)</AssemblyTitle>
    <Description>BlazarTester</Description>
    <Company>Mapromar</Company>
    <Authors>$(Company)</Authors>
    <Copyright>Copyright © 2005-2019 Mapromar</Copyright>
    <NeutralLanguage>en-CA</NeutralLanguage>
    <Version>0.0.0</Version>
    <FileVersion>0.0.0</FileVersion>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\Blazar.Core\Blazar.Core.bproj" />
    <ProjectReference Include="..\Blazar.System.Memory\Blazar.System.Memory.bproj" />
    <ProjectReference Include="..\Blazar.System.Console\Blazar.System.Console.bproj" />
    <ProjectReference Include="..\Blazar.System.Runtime\Blazar.System.Runtime.bproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Mapromar.Build.ProjectDependency" Version="1.*">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>
</Project>
```

Figure 21 Blazar project file.

DEBUGGER

Support is provided for debugging within Visual Studio, including breakpoints and inspection of variables and runtime state.

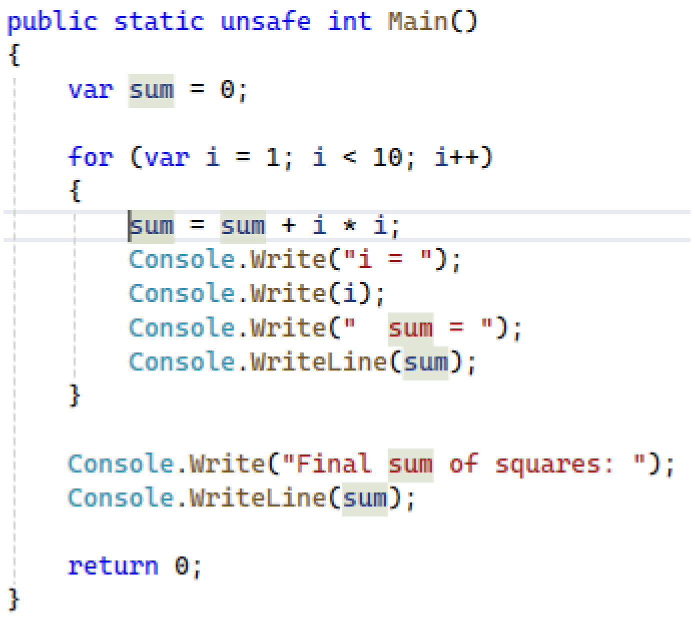


```
var sum = 0;

for (var i = 1; i < 10; i++)
{
    sum = sum + i * i;    ≤ 1ms elapsed
    Console.            ");
    Console.Write(i);
    Console.Write("  sum = ");
    Console.WriteLine(sum);
}
```

Figure 22 Halted program execution at breakpoint with variable inspection.

SYNTAX HIGHLIGHTING

Highlights all references to a variable under the cursor in the source code editor.

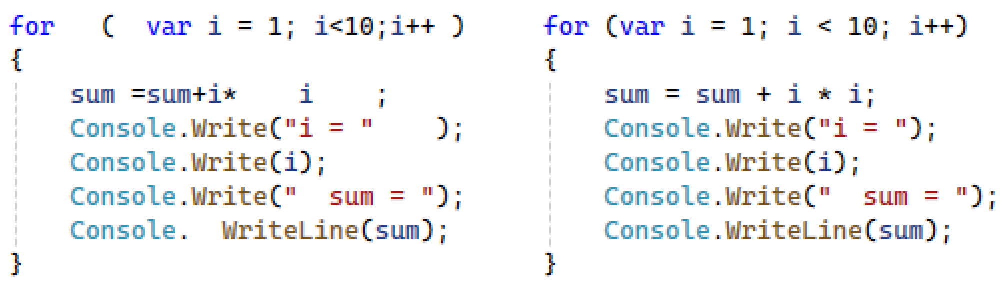


```
public static unsafe int Main()
{
    var sum = 0;

    for (var i = 1; i < 10; i++)
    {
        sum = sum + i * i;
        Console.Write("i = ");
        Console.Write(i);
        Console.Write("  sum = ");
        Console.WriteLine(sum);
    }

    Console.Write("Final sum of squares: ");
    Console.WriteLine(sum);

    return 0;
}
```

Figure 23 Highlighting all references to sum.

CODE FORMATTING

Formats whitespace and indentation in the source code editor to maintain consistent, standard coding practices.



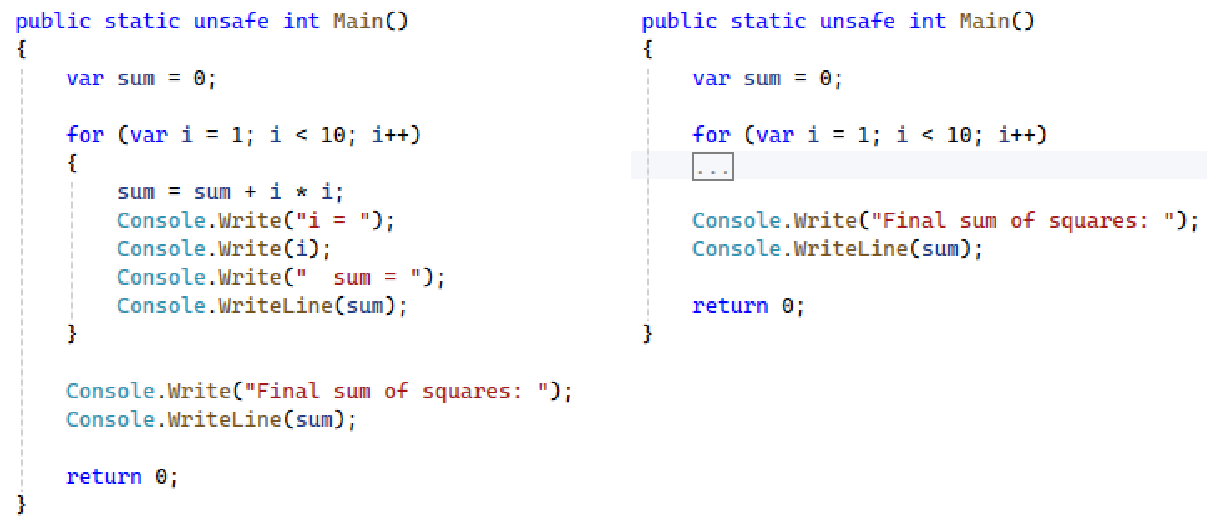
```
for   (  var i = 1; i<10;i++ )
{
    sum =sum+i*    i    ;
    Console.Write("i = "    );
    Console.Write(i);
    Console.Write("  sum = ");
    Console.  WriteLine(sum);
}
```

```
for (var i = 1; i < 10; i++)
{
    sum = sum + i * i;
    Console.Write("i = ");
    Console.Write(i);
    Console.Write("  sum = ");
    Console.WriteLine(sum);
}
```

Figure 24 Before and after code formatting.

Code Outlining

Allows code blocks in the source editor to be expanded or collapsed.



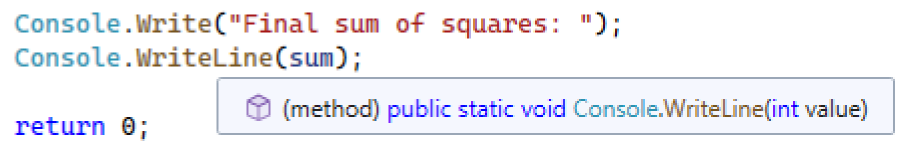
```
public static unsafe int Main()
{
    var sum = 0;

    for (var i = 1; i < 10; i++)
    {
        sum = sum + i * i;
        Console.Write("i = ");
        Console.Write(i);
        Console.Write("  sum = ");
        Console.WriteLine(sum);
    }

    Console.Write("Final sum of squares: ");
    Console.WriteLine(sum);

    return 0;
}
```

```
public static unsafe int Main()
{
    var sum = 0;

    for (var i = 1; i < 10; i++)
    ...

    Console.Write("Final sum of squares: ");
    Console.WriteLine(sum);

    return 0;
}
```

Figure 25 Expanded and collapsed for loop.

Signature Helper

Displays method modifiers, return type, and parameter names and types in a popup within the source editor.

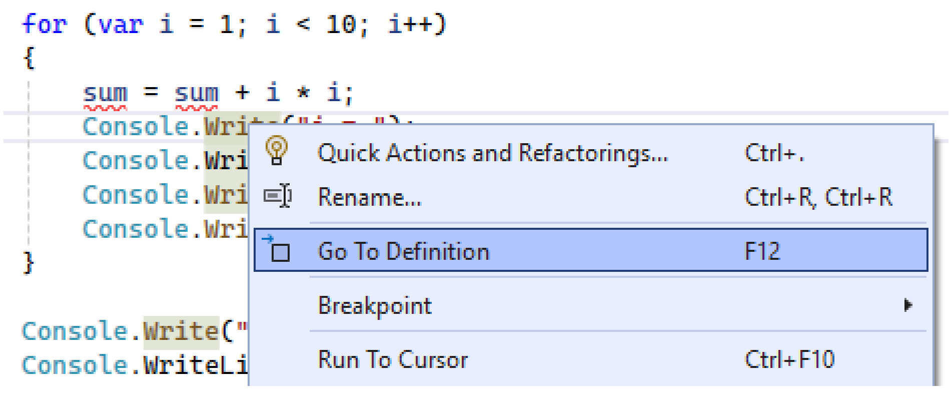


```
Console.Write("Final sum of squares: ");
Console.WriteLine(sum);

return 0;        (method) public static void Console.WriteLine(int value)
```

Figure 26 Signature for Console.WriteLine.

Go To Definition

Navigates to the location where an element is defined and positions the cursor at its declaration in the code file.

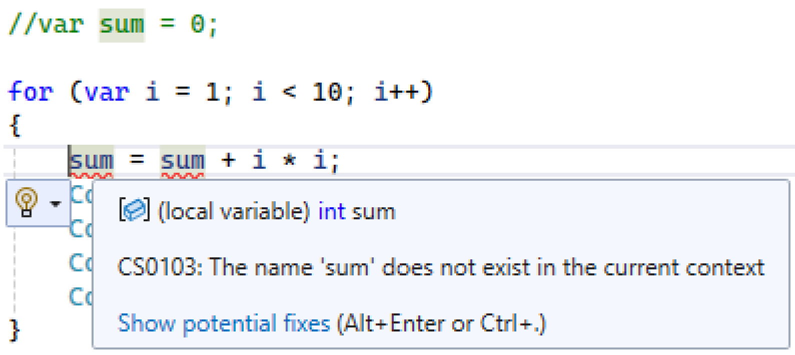


```
for (var i = 1; i < 10; i++)
{
    sum = sum + i * i;
    Console.Wri
    Console.Wri
    Console.Wri
    Console.Wri
}

Console.Write("
Console.WriteLi
```

Figure 27 Go to definition menu item.

Error Diagnostics

Error diagnostics display popups in the editor to indicate syntax errors in the source code.



```
//var sum = 0;

for (var i = 1; i < 10; i++)
{
    sum = sum + i * i;
    Co
    Co
    Co
    Co
}
```

(local variable) int sum

CS0103: The name 'sum' does not exist in the current context

Show potential fixes (Alt+Enter or Ctrl+.)

Figure 28 Error diagnostics for the variable sum.