SharpGenTools Documentation

Release 1.0.0

Jeremy Koritzinsky

Apr 22, 2022

Contents

CHAPTER 1

What is SharpGenTools

SharpGenTools is a code generator that generates C# for interoperation with C++ and COM libraries. It generates efficient C# code and handles all of the native marshalling. Additionally, it includes an assembly patcher to use CLR instructions not available from C# to further minimize the overhead of the generated code.

CHAPTER 2

Why SharpGenTools?

SharpGenTools makes it extremely easy for your C#-C++ interop to have "one source of truth", the C++ header files. Since SharpGenTools generates the C# headers on build by default, it helps ensure that they stay up to date with the matching C++ header files. It also directly integrates into MSBuild, so the user has do to minimal work to integrate it into their project. Also, all of the structures and interfaces are generated as `partial` structures/classes/interfaces, so you have the ability to fully customize them to your needs.

2.1 SharpGenTools vs CppSharp

Pros for SharpGenTools:

- Configuration file based
  - Does not require the user to build a driver program to generate C# code.
  - Generated code extremely configurable
  - Escape hatches to allow the user to manually write mappings SharpGenTools cannot handle.
- Full MSBuild integration
- Consumer file passthough
  - Consuming projects that also generate C# code will automatically gain knowledge about type mappings and defined types from project references and directly referenced packages.
- More efficient method calls on virtual methods of native instances (`calli` instruction vs. `Marshal.GetDelegateForFunctionPointer`).

Pros for CppSharp:

- More supported C++ features (operator overloading)
- Better support for custom passes, so may allow more configurability than SharpGenTools.

CHAPTER 3

Getting started

Check out *Getting Started with SharpGenTools* to get started!.

3.1 Release Notes

3.1.1 vNext.0

Features:

- Implemented default mapping for C++ `long`.
- Implemented support for different generation based on various platform ABIs with one build.
- Implemented support for Itanium ABI vtable offsets for single inheritance.

3.1.2 1.2

Features:

- Implemented relation mapping.
- Enabled automatic shadow generation for callbacks with array parameters when users specify length relations for array parameters.
- Emit `GC.KeepAlive` in marshalling cleanup to keep interface instances alive across calls.
- Use `__has_include` in generated headers to make SharpGen C++ generation/parsing process more deterministic.
- Update CastXML to be based off Clang 7.
- Added include files to MSBuild update checks so editing the native code that is being mapped will trigger a regeneration.

Bug Fixes:

- Fix NRE when VS is not installed.

3.1.3 1.1.2

Bug Fixes:

- Account for null struct-to-class instances when marshalling from native.

3.1.4 1.1.1

This patch fixes a number of bugs in 1.1 as well as refactoring the marshalling code-gen again to be more mantainable.

Features:

- Allow the user to specify extra arguments to pass to CastXML as a `@(CastXmlArg)` MSBuild item.

Bug Fixes:

- Pointer-sized members are correctly marshalled as return values.
- Ensure that `<short>` tag naming rules work on mapped constants.

Development Changes:

- Marshallers are now separated based on the conditions in which they are used. This makes them much easier to mantain.

3.1.5 1.1

This release focuses on improving codegen of already supported features and autogenerating some code that previously required manual generation.

Features:

- Flow information about callbacks to consuming projects.
- Opt-in autogeneration of shadows for callback interfaces.
- Miscellanous bug fixes.
- `ShadowContainer` initialization is thread-safe.
- Interface members of structs now causes native struct generation.
- SourceLink is now enabled in the package PDBs.

Bug Fixes:

- Miscellanous bug fixes for marshallers.
- `ShadowContainer` correctly adds shadows to the container when initialized.

Development Changes:

- Building now requires CMake and doesn't require the Windows SDK.
- The COM support library is now in a separate repository.
- Debug build now collects code coverage, enabling verification that our code works in our automated tests.
- MSDN Documentation provider and tasks in a separate repository.

3.1.6 1.0.0

This is the first stable release of SharpGenTools. Below is a summary of features available in SharpGenTools.

- Support for mapping C++ "interfaces"
- Support for generating C++ interface callbacks so C++ interfaces can be implemented in C#.
- Support for mapping C++ structures and auto-generating marshal types.
- Support for mapping C++ functions with C linkage.
- Custom doc providers supported.
- MSDN Doc provider in separate assembly.
- External documentation files support.
- Runtime support via SharpGen.Runtime package.
- Pre-mapped COM support via SharpGen.Runtime.COM package.
- Package dependency mapping flow - Include prologs, defines, type bindings, and doc links flow to consumers.
- Support for re-patching signed assemblies.
- Automatically locate the Visual C++ includes folder when including the standard library on Windows.

3.2 Getting Started with SharpGenTools

3.2.1 Installing SharpGenTools

To use SharpGenTools, you'll want to install the SDK package, as well as the Runtime package. The SDK package provides MSBuild tooling, and the runtime package provides the required support classes for the code generated from the SDK. You can use the following commands from the .NET CLI to install them:

```
dotnet add package SharpGenTools.Sdk
dotnet add package SharpGen.Runtime
```

If you want the support classes for COM libraries, you should also install the `SharpGen.Runtime.COM` package.

Note: These packages are separate so advanced and legacy users, specifically the SharpDX project, can use their own runtime support classes. In nearly all cases, you will want to reference both projects.

3.2.2 Making a Basic Mapping File

To generate C# using SharpGenTools, you have to create a mapping file.

Create a file named `Mapping.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<config id="MyMapping" xmlns="urn:SharpGen.Config">
    <assembly>MyAssembly</assembly>
    <namespace>MyAssembly.Namespace</namespace>
    <depends>SharpGen.Runtime</depends>
</config>
```

Let's go through each of these elements.

- The `config` tag: This is the root tag for your configuration file. The `id` attribute uniquely identifes your config file during your build.
- The `assembly` tag: Identifies what assembl for which you are generating code.
- The `namespace` tag: Identifies the root namespace for code generated from this config file.
- The `depends` tag: Declares dependencies on other config files. Optional, but can help ensure that all dependencies are correctly loaded.

Specifying Include Directories

Now that we have a mapping file, we can start declaring what C++ files to map.

To include a C++ file, we need to specify our include directories. There are two ways to specify include directories.

- **The <sdk> element:**
  - The `<sdk>` element can be used to automatically include common libraries, such as the C++ Standard Library and the Windows SDK. Some examples are below:

```
<sdk name="StdLib" />
<sdk name="WindowsSdk" version="10.0.15063.0" />
```

- **The <include-dir> element:**
  - The `<include-dir>` element allows you to specify a specific directory. Additionally, you can use the `$(THIS_CONFIG_PATH)` variable to refer to headers relative to the path to the config file.
  - The `override` member can be used to treat the headers as user headers instead of system headers.

```
<include-dir override="true">$(THIS_CONFIG_PATH)/../path/to/headers</include-
↪dir>
```

Including a C++ Header

Now that we've specified our include directories, we can actually include C++ headers in our code generation.

To include a header, we use the `<include>` tag. Below are a few examples:

```
<include file="header.h" namespace="MyAssembly.MyNamespace" attach="true" />
<include file="header1.h" namespace="MyAssembly.MyNamespace.SubNamespace" output=
↪"SubNamespace">
    <attach>MyType</attach>
    <attach>MyType2</attach>
    <attach>MyFunction</attach>
</include>
```

The `file` attribute specifies which file to include, and the `namespace` attribute specifies which namespace the C# elements generated from the C++ in this file should go in. The `output` attribute specifies what folder this include's namespace is output to. The `output` attribute has to be supplied on at least one `<include>` element for each sub-namespace. If it is applied multiple times, the last value takes effect.

Attaching Includes

You may have noticed above the `attach` attribute and the `<attach>` elements. These elements specify what C++ elements to actually generate C# interop for. If the `attach` attribute is set to `true`, all C++ elements in that include that SharpGenTools can map will be mapped. Alternatively, you can use `<attach>` elements in the include element to specify specific C++ elements to map. If neither is specified, no code is generated for any of the elements defined in that header. This allows you to specify headers needed for compilation even though they may not be needed for the mapping itself.

> **Warning:** For the both the `attach` attribute and the `<attach>` element, the C++ elements must be directly defined in that include file.
>
> Additionally, the name in the `file` attribute must match case with the first time the header was included, even if the header was first included transitively via a different header. If they don't match, the elements in the header will not be attached to the model.

3.2.3 Adding the Mapping File To the Build

Now that we have a basic mapping file, all we need to do is add it to the build!

In your `.csproj` file, add the line below:

```
<SharpGenMapping Include="path/to/Mapping.xml" />
```

SharpGenTools will now pick up your mapping file and generate C# for the C++ your config file specifies using the default mappings.

Note: The default mapping does not support mapping free functions. To map free functions, see the *Customizing your SharpGen Mapping* tutorial.

3.3 Customizing your SharpGen Mapping

3.3.1 Declaring C# types to SharpGen

SharpGen mantains an internal record of types it understands. If you want to reference a type it does not know about, then you have to declare that type to SharpGen. To declare a type to SharpGen, you use `<define>` or `<create>` elements under the `<extension>` element in the `<config>` tag. The following subsections show how to declare different types to SharpGen.

Note: Any types defined with the `<define>` tag will **not** have any code generated for them. SharpGen assumes that any types defined with the `<define>` tag already exist, either in source or in a referenced assembly.

Groups (Classes)

Groups represent a type for SharpGen to place functions and constants and other C++ constructs that cannot be represented in the C# object oriented model. You need to declare at least one group if you want to map any non-member functions with SharpGen. To map a group, use the following syntax:

```
<create class="MyNamespace.Functions" visibility="public static" />
```

The example above creates a `static` class named `Functions` in the `MyNamespace` namespace that is `public`. You can attach functions (shown in *Mapping Functions*) and constants (shown in *Macros and GUIDs as Constants*) to this class.

Interfaces/Classes

Interfaces represent C++ classes or structures with at least one pure virtual method defined. They are generated as classes normally, but can be configured to generate as interfaces (so C# classses can implement them and be passed to native code) with more advanced mappings. These interfaces are called callbacks. You can find more information about callbacks in *SharpGen Callbacks and Shadows*.

You will likely only define an interface if you are going to use it in a binding (see *Defining Default Type Bindings*). You can define an interface as shown below:

```
<define interface="Qualified.Name.For.MyInterface" />
```

There are a few other attributes for interface definitions. These are listed below.

- **native**
  - The native implmentation of the interface.
- **shadow**
  - The name of the shadow type of the interface. Will be used if any derived interfaces are callbacks. See *SharpGen Callbacks and Shadows* for more details.
- **vtbl**
  - The name of the vtbl type of the interface. Will be used if any derived interfaces are callbacks. See *SharpGen Callbacks and Shadows* for more details.

Structs

Structs represent any structures in C++ that do not have any virtual functions. If a struct is hard for SharpGen to model or you already have accurate marshalling for it in source or a referenced assembly, you can use a `<define>` tag to define it. For the most efficient and accurate code generation, you should calculate the size of the structure in bytes. Below is a simple example of defining a struct.

```
<define struct="Qualified.Name.For.MyStruct" sizeof="size_in_bytes" />
```

There are a few other attributes for struct defintions. These are listed below.

- **align**
  - The alignment of the structure. This generates as the `Pack` property on the `StructLayout` attribute.
- **marshal**
  - Whether or not this structure has a native marshal type. Because SharpGenTools uses the `calli` instruction, it has to handle native marshalling manually. If your type cannot be directly represented in unmanaged code from the managed instance, you must define marshalling code and set this attribute to `true`. See *SharpGen Native Marshalling* for documentation on how the SharpGenTools marshalling system works.
- **static-marshal**

  - Whether or not the marshalling functions are defined as static functions or instance functions. More information can be found at *SharpGen Native Marshalling*.

- **custom-new**
  - Whether or not there is a custom `__NewNative` method that should be used instead of just creating a new instance of the native structure. More information can be found at *SharpGen Native Marshalling*.

Enums

Enums represent a C++ enum. You would use a `<define>` tag for an enumeration that does not exist in code, but the parameter of a method can only take a finite subset of integer values. In that case, it would be helpful to your users to define a C# enum to ensure they only pass correct values. Below is an example for defining an enum:

```
<define enum="Qualified.Name.For.MyEnum" underlying="System.UInt32" />
```

The `underlying` attribute defines what type this enumeration is pretending to be. It does not need to match the declared underlying type in C#.

Additionally, there you can specify the `sizeof` attribute instead of the `underlying` attribute if you prefer to specify the enumeration in terms of the size of its native representation. See the table below for what underlying type is picked for each value of `sizeof`. If the `sizeof` value is not in the table, SharpGen will fail to generate code for your mapping.

`sizeof` value	Underlying Type
1	`System.Byte`
2	`System.Int16`
4	`System.Int32`

3.3.2 Defining Default Type Bindings

Sometimes the code for a specific type and the way it is used in the native code is hard for SharpGenTools to understand. In other cases, the type already exists in .NET and you want specific native types to always use it. You can use `<bind>` elements (in a `<bindings>` tag) to "bind" a native type to a managed type. An example of a bind element from SharpGen.Runtime is available below:

```
<bind from="int" to="System.Int32" />
```

Additionally, if you want the type to be represented by one type for the user but marshalled to native with a different type, you can set the `marshal` attribute on the `<bind>` element as shown below:

```
<bind from="bool" to="System.Boolean" marshal="System.Byte" />
```

In the example above, any element with type `bool` will be presented to users as a `System.Boolean`, but will be marshalled to and from native code as a `System.Byte`.

3.3.3 Mapping Functions

To map functions, you have to specify a group to which to attach the functions. Additionally, this group must be declared as shown in *Groups (Classes)*. You can use a `<map>` element under a `<mapping>` tag in the `<config>` tag. You use the `group` attribute to specify which group to attach the functions to. For example:

```
<map function="MyFunction" group="MyNamespace.Functions" />
```

3.3.4 Mapping Macros as Enums

In some C++ libraries such as DirectX, a set of macros define the valid values for a parameter. SharpGenTools allows you to map these macros into C# as an enumeration. To do so, you use a `<create-cpp>` element in the `<extension>` element as shown below.

```
<create-cpp macro="MY_MACRO_OPTIONS_.*" enum="MY_MACRO_OPTIONS" />
```

As you can see, you can use a regular expression in the `macro` attribute to select multiple macros to be members of this enumeration. This enumeration will be created during the parsing process and then mapped as a C++ enumeration with the default mapping rules.

3.3.5 Macros and GUIDs as Constants

Constants are mapped with the `<const>` tag under the `<extension>` tag. You can map both macros and GUIDs to constant values in the generated code. All constants need to be attached to a group or other type. Set the `class` attribute to the name of the class or type to which the constant should be attached to in the generated code. Set the `name` attribute to the name of the generated member in the C# code.

Mapping GUIDs

To map a GUID, set the `from-guid` attribute on the `<const>` tag to the name of the GUID in the C++ code.

Mapping Macros

Mapping macros to constants is more interesting. Since C++ macros do not inherently have a type, you gain a lot of control for how the macro is mapped as a constant. Below is a list of the different attributes you use to map macros.

- `from-macro`
  - Which macro you are mapping to a constant.
- `type`
  - The C# type of the constant.
- `cpp-type`
  - The C++ type of the constant. This is optional if it is the same as the C# type.
- `cpp-cast`
  - The type to cast the macro value to in C++. This is only needed if an explicit cast is needed for the literal macro value to be assigned to the C++ type of the constant.

Constant Value Mapping

For the `value` attribute of the `<const>` tag, you can specify any (HTML-escaped) C# expression. The placeholders in the following table allow you to substitute in information about the constant.

Placeholder	Subsititution
$0	C++ name of the macro or GUID
$1	Value of the macro or GUID
$2	C# name of the macro or GUID converted to Pascal Case
$3	Namespace declared in `<namespace>`

Note: The value of a GUID (for the placeholder $1) is the value of the result of a standard `ToString()` call on a `System.Guid` instance with the value of the mapped GUID.

3.3.6 Removing Elements

Sometimes you might want to map many elements in an include file, but not all of them. We supply the `<remove>` tag for you to remove items. Just set the attribute that matches the type of element you want to remove (from the table below) to a regular expression that matches all of the elements of that type you want to remove.

Element type to Remove	Attribute Name
Enum	`enum`
Enum Item	`enum-item`
Interface	`interface`
Method	`method`
Struct	`struct`
Field	`field`
Any other element	`element`
Multiple types of elements	`element`

Warning: Removing some elements (such as parameters or struct fields) will likely cause invalid marshalling code-gen and may destabilize your application.

3.4 Advanced Mappings

SharpGenTools allows you to heavily customize the mappings from C++ to C#. This page covers all of the currently supported mappings for each element type.

All attributes referred to below are on a `<map>` element under the `<mapping>` tag in the `<config>` tag.

Warning: Mapping rules in `<map>` tags apply to all matching elements in the mapping, not just the elements from includes in the current config file. This allows users with multi-assembly mapping and config files (described in *Advanced SharpGenTools Configuration Options*) to reuse mapping rules without having to include unneeded files in the reused config. You can limit what elements are matched to specific include files with context rules, explained in *Context Directives*.

3.4.1 Selecting Elements

Each mapping rule has to select elements to modify. You select elements by using a specific attribute on the `<map>` tag (see the *table below*), along with a C# regular expression as the value. Do note, the start and end matchers `^` and `$` are implied, so you do not need to add them to your regular expression.

To select an element, your regular expression needs to match its C++ qualified name. For enumeration items, you use the name that does not include the name of the enumeration. For methods, parameters, and fields, you use the qualified name of the parent concatenated with `::` and then the name of the element. For purposes of a "parent", the method is the parent of the parameter.

Additionally, you can prepend a `#` to your selector to select the immediate parent of the matched element. This is useful in mapping some COM libraries where an enumeration member is well known, but the enum name is a non-user-friendly generated name.

Mapping Tag Attribute	Elements Modified	Link to Specific Mapping Rules
`enum`	Enumerations	*Enums*
`enum-item`	Enumeration Items	*Enum Items*
`struct`	Structs	*Structs*
`field`	Struct Fields	*Fields*
`interface`	Interfaces	*Interfaces*
`method`	Interface Methods	*Functions and Methods*
`function`	Non-member Functions	*Functions and Methods*
`param`	Parameters	*Parameters*
`element`	All matching elements	All sections applicable
`doc`	Documentation Links	*Doc Links*

3.4.2 Context Directives

Context directives allow you to limit which includes mapping rules apply to. You can do this via the `<context>` tag.

To add a context for a set of mapping rules, add a `<context>IncludeName</context>` before the rules, and a `<context-clear />` after the rules. You can add multiple `<context>` tags to enable rules to affect a multiple headers and only those headers.

If you have multiple headers you want to repeatedly use as a singular context, you can define a context set such as below:

```
<context-set id="common-context">
    <context>firstHeader</context>
    <context>secondHeader</context>
</context-set>
```

You can then enable this context with the context tag as follows `<context>common-context</context>`.

If you want to limit rules to applying to C++ elements that were generated from a macro or GUID, you need to add two context tags as shown below:

```
<context>myConfigId</context>
<context>myConfigId-ext</context>
```

Warning: Additionally, the name in the `<context>` tag must match case with the first time the header was included, even if the header was first included transitively via a different header. If they don't match, the mapping rules will not affect the correct context.

3.4.3 General

- `name-tmp`
  - Rename the element to the name given here and allow naming rules (in *SharpGen Naming Rules*) to still run on these elements.
  - The value here can be a C# regex replacement expression for the corresponding regex used in the selector.
- `name`
  - Rename the element to the name given here and do not allow naming rules (in *SharpGen Naming Rules*) to still run on these elements.
  - The value here can be a C# regex replacement expression for the corresponding regex used in the selector.
- `visibility`
  - Override the visibility and other modifiers of the resulting C# element. Options are listed below.
  - public
  - internal
  - protected
  - private
  - override
  - abstract
  - partial
  - static
  - const
  - virtual
  - readonly
  - sealed
- `naming`
  - Override default naming rules. Options are listed below and explained in more detail in *SharpGen Naming Rules*
  - `default`
    - Use default naming rules.
  - `noexpand`
    - Do not expand short name rules for the name of this element.
  - `underscore`
    - Keep the underscores between each part of the original name if it was `snake_case`.

3.4.4 All Types

- `assembly`
  - The assembly the type should be in. When specified, this requires a `namespace` attribute as well.
- `namespace`
  - The namespace the type should be in. When specified, this requires an `assembly` attribute as well.

3.4.5 All Marshallable Elements

- `type`
  - The type presented to the user in the mapping.
- `override-native-type`
  - Override the native representation with the type specified in the `type` attribute.
- `pointer`
  - Override the pointer arity of the matching C++ elements.
- `array`
  - Override the array dimension of the matching C++ elements.
- `relation`
  - Specify that this marshallable element is related to another marshallable element or has a constant value. See *SharpGen Mapping Relations* for more information.

3.4.6 Enums

- `flags`
  - Specifies if this enumeration should be generated with the `[Flags]` attribute.
- `none`
  - Specifies if SharpGenTools should generate a member named None with the value 0.

3.4.7 Enum Items

Enum items can only be configured with the *General* rules.

3.4.8 Structs

- `native`
  - Force generation of native marshal type for this struct.
- `struct-to-class`
  - Generate this structure as a C# `class` instead of a `struct`.
- `marshal`

  - Marks this struct as having custom marshalling methods, so marshalling methods will not be generated. See *SharpGen Native Marshalling* for details.

- `static-marshal`
  - Marks that this struct uses static marshalling methods. See *SharpGen Native Marshalling* for details.
- `new`
  - Marks that this struct's native structure has a custom construction method instead of the constructor. See *SharpGen Native Marshalling* for details.
- `marshalto`
  - Forces generation of the `__MarshalTo` method. See *SharpGen Native Marshalling* for details.
- `pack`
  - Specifies the packing/alignment of the structure.

3.4.9 Fields

Fields can only be configured with the *General* and *All Marshallable Elements* rules.

3.4.10 Interfaces

- `callback`
  - Generate this interface as a callback interface. See *SharpGen Callbacks and Shadows* for details.
- `callback-dual`
  - Generate this interface as a callback interface, but also generate a default implementation for C++ instances of the interface. See *SharpGen Callbacks and Shadows* for details.
- `callback-visibility`
  - The visibility for the default implementation. See the *visibility options* for possible values.
- `callback-name`
  - A custom name for the default implementation. Defaults to the original name + "Native".
- `autogen-shadow`
  - Automatically generate the shadow classes for this callback interface. See *SharpGen Callbacks and Shadows* for details.
- `shadow-name`
  - A custom name for the shadow type. See *SharpGen Callbacks and Shadows* for details.
- `vtbl-name`
  - A custom name for the vtbl type. See *SharpGen Callbacks and Shadows* for details.

3.4.11 Functions and Methods

- `check`
  - Enable or disable automatically checking the error code. Defaults to true (enabled).
- `hresult`

  - Force the error code to be returned.

- `rawptr`
  - Force generation of a private overload of the method that has all array, pointer, or interface parameters as `IntPtr`.
- `return`
  - Always return the return value of the function.
- **`type`**
  - The user-visible return type of the element.

Rules for methods only:

- `property`
  - Enable or disable automatic property generation. Defaults to true (enabled).
- `persist`
  - Cache the value for the generated property getter the first time the getter is called.
- `custom-vtbl`
  - Generate a private variable for the virtual method table index of the method, so it can be customized at runtime as needed.
- `offset-translate`
  - Offset the virtual method table index by a this value.
- `keep-implement-public`
  - If the parent interface has `dual-callback` set to `true`, then keep the implementation of this method in the default implementation public.

Rules for functions only:

- `dll`
  - The expression to put in the `DllImport` attribute as the dll name.
- `group`
  - The group (see *Groups (Classes)*) to attach the function to.

3.4.12 Parameters

- `attribute`
  - Override the attributes for this parameter. Options are below:
  - `none`
  - `in`
    - This parameter is an input parameter.
  - `out`
    - This parameter is an output parameter.
  - `inout`
    - This parameter is both an in and out parameter;

  - `buffer`
    * This parameter takes in an array of elements.
  - `optional`
    * This parameter is optional.
  - `fast`
    * If this parameter is an out parameter, reuse the C# interface instance for the returned value by setting the `NativePointer` property.
  - `params`
    * Use the C# params modifier on this parameter.
  - `value`
    * Force a C# value type to be passed by value to the generated method for this parameter even if the size is greater than 16 bytes.
- `return`
  - Use this parameter as the return value of the generated C# method/function for this parameter's parent C++ method/function.

3.4.13 Miscellaneous

Doc Links

- `name`
  - The element that all doc references to the selected elements should reference in the generated documentation.

3.5 SharpGen Mapping Relations

Sometimes fields and parameters have values that are related to the containing structure (for fields), a constant value, or the length of another array field or parameter. Instead of either requiring your users to specify these values or requiring you to wrap the SharpGen-generated code in your own interface, SharpGen supports defining these relationships in your mapping files. SharpGen will then generate code that implements these relations so you do not have to wrap the code yourself. When the relation rules are applied, the field with the relation rule is hidden from the managed view of the structure, method, or function and is calculated when marshalling.

3.5.1 Struct Size Relation

A common pattern in Windows APIs is to have a field, usually named `cbSize` that must be assigned the size of the structure. So, SharpGen provides a relation to automatically calculate this field for you. The following rule tells SharpGen that all fields named `cbSize` must be assigned the size of their containing structure.

```
<map field=".*::cbSize" relation="struct-size()" />
```

All of the `cbSize` fields in a struct's native representation will be assigned the size of their containing (native) structure when marshalling the structure to its native representation.

This relationship is only valid on fields.

3.5.2 Constant Value Relation

Sometimes APIs have "reserved" fields or parameters, parameters that must have a specific value and are reserved for later use or undocumented. Since these fields and parameters will always have the same value, there is no value in requiring your consumer to supply these values. So, by using the constant-value relation, SharpGen will supply the reserved values itself when marshalling. For example, let's say that the function `Foo` takes a parameter named `ireserved` that is required to always have the value `0`. The mapping rule below will instruct SharpGen to always assign the value `0` to the `ireserved` parameter.

```
<map param="Foo::ireserved" relation="const(0)" />
```

This relation is valid on both fields and parameters.

3.5.3 Length Relation

When passing an array to native code, it is common to pass the length of the array as a separate parameter. Additonally, when generating a shadow implementation for an interface, SharpGen must know how long to construct the managed view of array parameters. To convey this information, SharpGen provides a relation for array length. For example, if a function `Bar` takes two parameters `arr` and `len` where `len` is the length of `arr`, then the following rule will tell SharpGen that `len` is the length of `arr`.

```
<map param="Bar::len" relation="length(arr)" />
```

This relation not only automatically calculates the value of the `len` parameter when calling the native function, but when applied to a method, it allows SharpGen to correctly automatically generate a callback implementation when generating a shadow. Additionally, for managed to native calls, if a parameter has a native property named `Length`, this relationship can be defined to set a parameter to have the value of that `Length` property.

This relation is valid on both fields and parameters.

3.6 SharpGenTools Config File Features

The SharpGenTools config files have a few nice features to ease mapping development.

3.6.1 Macros

You can define SharpGen macros in your project file as below:

```
<PropertyGroup>
    <SharpGenMacros>$(SharpGenMacros);MY_MACRO</SharpGenMacros>
</PropertyGroup>
```

Then, within your config file, you can surround **any** XML tag(s) with an `<ifdef name="MY_MACRO">` tag. Then, when `MY_MACRO` is defined, those rules/includes will be included, but otherwise they will be ignored.

3.6.2 Variables

Variables can be defined in your mapping file to give you some shorthand to use common expressions. You can define a variable with the `<var>` tag as shown:

```
<var name="varname" value="value" />
```

You can refer to this variable in any attribute value or element value as `$(varname)`. The variable `$(THIS_CONFIG_PATH)` is to the directory path the file is in.

3.6.3 Dynamic Variables

Dynamic Variables allow your mapping rules to use the values of C++ macros in attributes or element values of mapping and binding rules. A dynamic variable is defined for every C++ macro SharpGenTools parsed while parsing the C++ headers. To use a dynamic variable, you can use the syntax `#(DYNAMIC_VARIABLE_NAME)`. SharpGen will give you an error if the dynamic variable cannot be found.

3.7 SharpGen Naming Rules

3.7.1 Default Naming Rules

SharpGen has a few default name mappings that it applies to C++ elements when creating the C# model. Below is an ordering of how SharpGenTools maps names:

1. If this C++ element has a name specified from a mapping rule, use that name.
2. If the mapping rule is not set using `name-tmp`, stop here.
3. If the name is not in `snake_case`, not in all caps, and the first letter is capitalized, stop here.
4. (Enum Items only) If the item name starts with the name of the enumeration, remove the enumeration name.
5. Remove the leading underscore if one exists.
6. Apply custom rules (explained in *Adding Naming Rules*) if the naming flags aren't set to `noexpand`.
7. Convert to Pascal case.
8. (Pointer parameters only) If the name originally started with `pp`, append `Out` to the final name and remove the `pp` prefix. If the name originally started with just `p`, append `Ref` to the final name and remove the `p` prefix.
9. (Parameters only) If the final name starts with a digit, prepend `arg` to the name.
10. (Parameters only) Convert to `camelCase`.

3.7.2 Adding Naming Rules

In addition to the direct name mappings and the default naming rules defined above, there are extension points in configuration files for custom naming rules. All of these rules go under the `<naming>` tag under the `<config>` tag.

"Short" Rule

"Short" rules are basic regex text substitution rules. You can use them to expand domain specific acronyms. You can add a short rule like below:

```
<short name="CLR">CommonLanguageRuntime</short>
```

3.8 Making SharpGenTools Document Your Mappings

SharpGenTools gives you two options for how to automatically document your mappings.

3.8.1 External Documentation Comments

You can supply external documentation though XML files structured as below:

```
<comments>
    <comment id="C++ or C# Element Name">
        <summary>Summary here</summary>
    </comment>
</comments>
```

SharpGenTools will automatically include an `<include>` documentation tag that points to a matching element in an external documentation file. You can specify an external comments file to SharpGen by adding it as a `SharpGenExternalDocs` item in your project file.

3.8.2 Doc Providers

Sometimes you're mapping an already documented library, and you don't want to have to manually extract the documentation for each element you're documenting. SharpGen provides the `IDocProvider` interface that you can extend:

```
#nullable enable

using System.Threading.Tasks;

namespace SharpGen.Doc;

/// <summary>
/// An <see cref="IDocProvider"/> implementation is responsible to provide␣
↪documentation to the Parser
/// in order to feed each C++ element with an associated documentation.
/// This is optional.
/// A client of Parser API could provide a documentation provider
/// in an external assembly.
/// </summary>
public interface IDocProvider
{
    /// <summary>
    /// Finds the documentation for a particular C++ item.
    /// </summary>
    /// <param name="fullName">
    /// The full name.
    /// For top level elements (like struct, interfaces, enums, functions),␣
↪it's the name of the element itself.
    /// For nested elements (like interface methods), the name is of the␣
↪following format: "IMyInterface::MyMethod".
    /// </param>
    /// <param name="context">Environment for documenting, used to create␣
↪items and subitems</param>
```

```
    /// <returns>Non-null documentation item container created by <see␣
↪cref="IDocumentationContext"/></returns>
    Task<IFindDocumentationResult> FindDocumentationAsync(string fullName,␣
↪IDocumentationContext context);

    /// <summary>
    /// If true, any exception thrown, or any error logged by this provider␣
↪will cause the build to fail.
    /// </summary>
    /// <remarks>
    /// For providers that rely on unstable factors (networking), it is␣
↪recommended to set this to <c>false</c>.
    /// However, the default choice should be <c>true</c>.
    /// </remarks>
    bool TreatFailuresAsErrors { get; }

    /// <summary>
    /// Name of the documentation provider to be presented to user when␣
↪needed.
    /// </summary>
    /// <remarks>
    /// Short version. Without words like "documentation provider" or␣
↪"extension".
    /// </remarks>
    string UserFriendlyName { get; }
}
```

You can reference the SharpGen package on NuGet to get a reference to the assembly. To enable installed doc providers, set the `$(SharpGenGenerateDoc)` property to `true`. By default, SharpGenTools.Sdk does not ship with any doc providers.

3.8.3 Doc Providers MSBuild Integration

To integrate into MSBuild, you need to create an MSBuild task. The MSDN Doc Provider task project in the SharpGen.Doc.Msdn repository (SharpGen.Doc.Msdn.Tasks), is a great example of how to create your own doc provider and hook it into the build process. If you are going to publicly publish the doc provider task, please make the task share the same condition statement as the one in SharpGen.Doc.Msdn.Tasks so users can easily enable the provider in a standard way.

3.9 SharpGen Native Marshalling

Because SharpGen uses `calli` instructions instead of `Marshal.GetDelegateForFunctionPointer` delegates, it bypasses all of the .NET runtime's built in marshalling code. As a result, SharpGen has to ensure that each type it passes to and from native code has the same bit-level representation as the native representation of the type.

3.9.1 Default Marshalling

By default, SharpGen only generates a native marshal type for types that have a different managed representation. The different conditions that trigger generation of a native marshal struct are listed below. If any condition is true for at least one field, a marshal struct is generated.

Array fields

Fields with fixed length arrays require a native marshal struct since the managed representation cannot lay out an array member in memory without forcing all consumers to use `unsafe`, which does not make a good user experience.

String fields

String fields are always marshalled across as a `System.IntPtr`. As a result, any string fields force generation of a marshal structure.

Interface fields

Interface fields are mashalled across as a `System.IntPtr` but presented in the user structure as an instance of the interface type.

Overridden User-facing Type

If the user-facing type is overriden without overriding the native type (via the `override-native-type` attribute on a mapping rule), then we need to generate a marshal struct to correctly marshal between the native representation and the managed structure.

Bind Marshal Type

If a type is bound with a `<bind>` rule and the `<bind>` rule has a `marshal` attribute that differs from the `to` attribute, then a marshal struct will be generated.

Struct fields with Marshal Structs

As should be obvious, if a member of this struct has a struct type that needs a native marshal struct, then we have to generate a native marshal struct for this struct as well.

Special Case: Enums

Enum types have the same bit-level representation as their underlying type, so we don't need to generate a marshal struct.

Special Case: Bool fields backed by Integer types

If an integer field is mapped to a bool, we can do all conversions efficiently with just the regular struct type while keeping the native representation of this field the same. So, for these fields, called "bool to int" fields, we don't need to generate a native marshal structure.

3.9.2 Marshalling Member Names

Here's a list of the names and signatures of members that SharpGen generates for marshalling. All of these members are by default `internal` instance members on the structure.

- `struct __Native`

  - The structure with the native representation of the struct.
- `void __MarshalFrom(ref __Native @ref)`
  - Marshals from the native struct to the managed struct.
- `void __MarshalFree(ref __Native @ref)`
  - Free any unmanaged resources, such as memory, that was allocated for the native structure.
- `void __MarshalTo(ref __Native @ref)`
  - Marshal data from the managed structure to the native structure. Only generated if needed.

3.9.3 Marshalling Extension Points

Custom Marshal Functions

When custom marshalling is enabled through a mapping rule, SharpGen will generate code assuming that the structure has native marshalling, but you as the consumer must manually write the marshalling methods. You use this setting when SharpGen can't correctly generate the native structure and the marshalling.

Static Marshal Functions

When static marshalling is enabled through a mapping rule, SharpGen will generate marshalling code in the structure, but methods will reference static marshalling methods with the following signatures that you must supply:

- `static void __MarshalFrom(ref StructName @this, ref __Native @ref)`
- `static void __MarshalFree(ref __Native @ref)`
- `static void __MarshalTo(ref StructName @this, ref __Native @ref)`

Custom Native New Function

Sometimes you might need to specially initialize some of the members of your structure when constructing it. As a result, SharpGenTools allows you to specify a custom function to create new instances of the native instance that will be used when constructing single instances (non-arrays) of the native structure. The signature you must supply is below:

```
static __Native __NewNative();
```

3.10 SharpGen Callbacks and Shadows

SharpGenTools has support as part of the SharpGen.Runtime package for inheriting from interfaces generated from C++ and passing C# implementations to native code. In the mapping files, the `callback` mapping rule specifies SharpGen should generate an interface instead of a class for the C++ interface.

3.10.1 Auto-Generated Shadows

To auto-generate the shadow for a callback, add the `autogen-shadow="true"` attribute to a mapping rule for the callback interface. This will automatically generate a shadow interface corresponding with the instructions below using the same marshalling code-gen infrastructure as the rest of the generated code.

If any parameters of any methods on the interface are arrays, you must provide an array length relation in your mapping rules. Otherwise your code will be generated with an unconditional `InvalidOperationException`. See *Length Relation* for more information.

> **Warning:** Property generation does not currently play nice with the shadow auto-generator. You may need to use a mapping rule to disable property generation. See *Functions and Methods* for details on the mapping rule.

3.10.2 Manually-Written Shadows

To declare the shadow type for a callback, put the `SharpGen.Runtime.ShadowAttribute` attribute on the interface type. The parameter is a `typeof(MyInterfaceShadow)` expression where the type is the shadow type. For an example, look at the code in `ComStreamBaseShadow.cs` in the `SharpGen.Runtime.COM` folder.

Note: SharpGen does not currently generate method signatures for callback interfaces if it is not auto-generating shadows. You will have to specify the method signatures in another file defining the interface.

Shadow Objects

The shadow object is the object that connects the native object with the .NET object instance. Most of your shadow classes will be similar to the shadow class below:

```
private class MyInterfaceShadow : BaseInterfaceShadow // Use CppObjectShadow if there␣
↪isn't a base interface
{
    private static MyInterfaceVtbl Vtbl = new MyInterfaceVtbl(0);

    protected override CppObjectVtbl GetVtbl => Vtbl;
}
```

Most of the interesting work happens in the Vtbl type, which we cover in the next section.

Virtual Method Table (Vtbl) Type

The Vtbl type builds the virtual method table and has methods to marshal calls from native back into the managed code. The Vtbl class should be declared similar to as follows within the Shadow type:

```
protected class MyInterfaceVtbl : BaseInterfaceVtbl // Use CppObjectVtbl if there isn
↪'t a base interface
{
}
```

Vtbl Constructor

The Vtbl constructor should look similar to below:

```
public MyInterfaceVtbl(int numberOfMethods)
    :base(numberOfMethods + numMethodsInMyInterface)
{
```

(continues on next page)

(continued from previous page)

```
}
```

The `numberOfMethods` parameter allows derived classes to also be callbacks. The base vtbl class requires the number of methods on the interface to correctly allocate for the native vtbl.

There are statements we need to put into the constructor, but we will cover those in the next section.

Vtbl Methods

Each method in the interface has to have a corresponding vtbl method and delegate. The delegate has to have an `[UnmanagedFunctionPointer]` attribute that specifies the native calling convention. Additionally, the delegate must have the first parameter be an `IntPtr` to represent the `this` pointer, and the remaining parameters must be directly representable directly in native code. To add it into the vtbl, add a statement in the constructor in the order that it was declared in native code:

```
AddMethod(new MyMethodDelegate(MyMethod));
```

The declaration of the method must be `static` and have the same parameters and return value as the delegate.

Guidelines for Vtbl Method implementations

Here are a few guidelines for vtbl method implementations:

- To get the shadow object for the `this` pointer, call `ToShadow<MyInterfaceShadow>(thisPtr)`.
- To get the .NET object from the shadow, cast the `Callback` property on the shadow object to the interface type.
- Wrap the call to the .NET object in a `try-catch` so .NET exceptions do not escape into native code. I make no promises that escaping exceptions won't crash the application.

3.11 SharpGen Runtime Platform Detection

Various platforms have differences in their native calling conventions or in the sizes of native types. To support these platforms, SharpGenTools supports transparently generating code that correctly handles multiple platforms in a single set of generated code. In SharpGenTools version 2.x and newer, the default-generated code will support all supported platforms if the code requires it.

3.11.1 Selecting Specific Platforms

Sometimes the native library that you are mapping is only available on a specific platform. For example, the DirectX suite is Windows-only. For these platforms, there is no reason to have code generated to support non-Windows platforms. You can request that your code is only generated to support specific platforms by adding the following elements to your project file:

```
<ItemGroup>
    <SharpGenPlatforms Include="Windows" /> <!-- Enable code-gen for Windows␣
↪platforms -->
```

(continues on next page)

(continued from previous page)

```
    <SharpGenPlatforms Include="ItaniumSystemV" /> <!-- Enable code-gen for non-
↪Windows platforms implementing the Itanium and SystemV ABIs (Linux, macOS, and␣
↪others) -->
</ItemGroup>
```

3.11.2 Selecting Specific Platforms with RIDs

In addition to using `<SharpGenPlatforms />` items, SharpGenTools supports inferring which platforms to generate code for by the `<RuntimeIdentifier />` property. As a result, self-contained .NET Core deployments or RID-specific framework dependent appliations will automatically only generate code for the specific platform they are compiling for.

3.12 Advanced SharpGenTools Configuration Options

SharpGenTools gives you many different configuration options in MSBuild. They're listed below:

- `SharpGenGenerateDoc`
  - Generate documentation from a doc provider (see *Making SharpGenTools Document Your Mappings*)
  - Defaults to `false`
- `SharpGenGenerateConsumerBindMapping`
  - Generate a bind mapping file that allows consumers of your project that use SharpGenTools to extend your mappings without having to copy yours.
  - Defaults to `true`
- `SharpGenGlobalNamespace`
  - Use the given namespace as the namespace for the runtime support types SharpGen uses.
  - Defaults to `SharpGen.Runtime`
- `SharpGenIncludeAssemblyNameFolder`
  - Include the name of the assembly as a directory in the output path.
  - Defaults to `false`
- `SharpGenDocumentationOutputDir`
  - The output directory for the documentation cache.
- `SharpGenGeneratedCodeFolder`
  - The name of the folder in the output path to place the generated code in.
  - Defaults to `$(IntermediateOutputPath)SharpGen/Generated`
- `SharpGenOutputDirectory`
  - The base directory for code output.
  - Defaults to `$(MSBuildProjectDirectory)`

3.12.1 Generated Code Output Directory Structure

Some of the options above control the output directory for generated code. The two options below are the different ways the output path is constructed.

- If `SharpGenIncludeAssemblyNameFolder` is false, `$(SharpGenOutputDirectory)/$(SharpGenGeneratedCodeFolder)`
- If `SharpGenIncludeAssemblyNameFolder` is true, `$(SharpGenOutputDirectory)/SharpGenAssemblyName/$(SharpGenGeneratedCodeFolder)`

3.12.2 CastXML Customization

- `CastXmlPath`
  - A path to a custom build of CastXML.
- `CastXmlArg` MSBuild Items
  - Additional arguments to pass to CastXML when parsing the C++ code.

3.13 Mapping Limitations

Below are a list of limitations in SharpGen's current generator. Many of these are on the roadmap for future versions.

- Does not map virtual but not pure virtual member functions.
- Does not map classes with virtual functions and state.
- Does not call constructors of C++ classes.
- Non-member functions must be declared `extern "C"` for the `DllImport` entry point to find the function.
- Does not map state for classes with pure virtual members.
- Does not map non-virtual member functions.
- Does not map function pointer parameters to C# delegate types.