orphan: Eventually, we would like to write Swift modules which define pure-C entry points for top-level functions, and be able to export more data types to C code.

This will be important for the Linux port, but also perhaps for system frameworks that want to transition to Swift.

The radars tracking this work are:

- rdar://22488618 - @c top-level functions
- rdar://22490914 - @c structs

The new @c attribute

This attribute can be applied to the following kinds of declarations:

- top-level functions
- static methods in non-generic classes
- enums
- structs

There are two forms of the attribute:

```
@c
@c(asmname)
```

The latter allows the exported name to be set. By default, the exported name is the unmangled, unqualified name of the function or nominal type.

There is the question of how to gracefully handle name conflicts inside a module. Since C does not have real modules or qualified names, we probably can't catch name conflicts until link time. At the very least, we will prohibit overloading `@c` functions (unless we use Clang-style mangling and `__attribute__((overloadable))`).

However, we might want to prefix the default @asmname of a @c symbol with the Swift module name followed by an underscore, instead of using the unqualified name.

Type bridging

The rules for bridging types in `@c` function signatures are a subset of `@objc`.

Bridgeable types are now partitioned into two broad categories, "POD" and "non-POD". POD types include:

- integers
- floating point numbers
- @c enums
- fixed size arrays (currently presented as homogeneous tuples of POD types)
- @c structs (whose fields must all be POD types)
- pointers to C types
- @convention(c) function types

On Linux, we can't have reference counted pointers here at all, and NSArray, etc do not exist, so only POD types are bridgeable. We must ensure that we produce the right diagnostic and not crash when the user references NSArray, etc on Linux.

On Darwin, we can allow passing reference counted pointers directly as function parameters. They are still not allowed as fields in `@c` structs, though.

The convention for arguments and results can be the same as CoreFoundation functions imported from C. The code in `CFunctionConventions` in SILFunctionType.cpp looks relevant.

Bridging header output

We can reuse most of `PrintAsObjC` to allow generating pure-C headers for Swift modules which use @c but not @objc.

Exporting functions to C

Applying `@c` to a function is like a combination of `@convention(c)` and `@asmname(func_name)`.

The types in the function signature are bridged as described above, and a foreign entry point is generated with the C calling convention and given asmname.

When the function is referenced from a `DeclRefExpr` inside of a `FunctionConversionExpr` to `@convention(c)`, we emit a direct reference to this foreign entry point.

@c applied to enums and structs

For enums, `@c` and `@objc` can be synonyms. We still have to track which one the user used, for accurate printing. On Linux, `@objc` could probably be changed to always diagnose, but this will require changing some tests.

As stated above, all the fields of a `@c` struct must themselves be POD.

Structs declared as `@c` need to be laid out with C size and alignment conventions. We already do that for Swift structs imported from Clang by asking Clang to do the layout on Clang AST, so perhaps for `@c` structs declared in Swift, we can go in the other direction by constructing Clang AST for the struct.

Accessibility and linkage for @c declarations

Only public enums and structs should appear in generated headers; for private types, `@c` only affects layout and restrictions on the field types.

For functions, it is not clear if private together with `@c` is useful, but it could be implemented for completeness. We could either give the foreign entry point private linkage, or intentionally give it incorrect linkage allowing it to be found with `dlsym()`.

inout parameters in @c and @objc functions

Right now we don't allow `inout` parameters for `@objc` methods. We could export them as nonnull pointers, using `__attribute__((nonnull(N)))` rather than `_Nonnull`.

If we ever get as far as implementing C++ interoperability, we could also export inouts as references rather than pointers.

Diagnostics

Right now all diagnostics for type bridging in Sema talk about Objective-C, leading to funny phrasing when using invalid types in a `@convention(c)` function, for instance.

All diagnostics need to be audited to take the language as a parameter, and either say `cannot be represented in C` or `cannot be represented in Objective-C`. A Linux user should never see diagnostics talking about Objective-C, except maybe if they explicitly mention `@objc` in their code.

On the plus side, it is okay if we conservatively talk about `C` in Objective-C diagnostics on Darwin.

Cleanups

Right now various aspects of the type bridging mapping are duplicated in several places:

- ASTContext::getBridgedToObjC()
- TypeChecker::isRepresentableInObjC() (various overloads)
- include/swift/ClangImporter/BuiltinMappedTypes.def
- include/swift/SIL/BridgedTypes.def
- TypeConverter::getLoweredCBridgedType()
- ClangImporter::VisitObjCObjectPointerType() and other places in ImportType.cpp
- PrintAsObjC::printIfKnownGenericStruct()
- PrintAsObjC::printIfKnownTypeName()

We should try to consolidate some of this if possible, to make the rules more consistent and easier to describe between Darwin and Linux.