Carbon Language - http://github.com/carbon-language

# Carbon: C++ interop for overloaded functions and function templates

**Authors:** zygoloid, chandlerc
**Status:** Done
**Created**: 2025-03-17
**Docs stored in Carbon's Shared Drive**
**Access instructions**

## Abstract

When calling a C++ overload set from Carbon, the rules that decide exactly which overload to call and how to initialize the parameters are shared between Carbon and C++, and the work to implement those rules are shared between the Carbon toolchain and Clang. The C++ side determines *what* is called, and the Carbon side determines *how* it is called.

## Background

Related docs:

- Carbon: Interop - Using C++: High-level design doc.
- Carbon: C++ Interop for constructors: Design for interop with constructors in particular.

### Overload sets

The basic unit of callable API in C++ is an *overload set*. This is a collection of functions and function templates that were found by name lookup as potential candidates for a call. An overload set should be treated as a unit and not divided into individual declarations, because overloads may misbehave if called with arguments that would have been a better match for another overload.

In modern C++, even a single non-templated function is considered to be an overload set, and uses the same rules as an overload set that can produce multiple different candidates.

### Implicit conversion sequences in C++ overload resolution

When overload resolution in C++ considers call candidates, it builds *implicit conversion sequences* to determine whether each argument can be converted to the corresponding

parameter, and approximately how that would be done. The implicit conversion sequences for each argument are ranked and compared across overload candidates to determine which overload should be called – the selected overload must have the best (or tied for best) implicit conversion sequence for every argument.

Implicit conversion sequences are a virtual representation of what an implicit conversion might do, and the rules for forming them mostly mirror the C++ implicit conversion rules. However, they are not identical to the implicit conversion rules:

- There are cases where an implicit conversion sequence exists but no implicit conversion exists, where the C++ rules believe they have identified the function that the developer "meant to" call, even though it is not actually callable. For example:
  - Given an argument that is a glvalue of type `T`, an implicit conversion sequence to a parameter of type `T` always exists and always has "identity" rank, even if `T` is not copyable or moveable.
  - Implicit conversion sequence formation typically does not take access into account. An implicit conversion sequence can be formed when the argument is of type `Derived*` and the parameter is of type `PrivateBase*`.

  If an overload candidate is selected for which implicit conversion sequences can be formed but implicit conversions cannot, the later process of building a call to the selected candidate will fail with an error.
- There are rare cases where an implicit conversion sequence does not exist despite an implicit conversion being possible. Initially this was assumed to not happen, and the C++ rules historically did not perform overload resolution when the only candidate was a single non-templated function, but the rules were changed to perform overload resolution even for that case so that, among other concerns, calls with no viable implicit conversion sequence would be rejected even if an implicit conversion is possible.

# Proposal

When an overloaded C++ function is called from Carbon, the overload is resolved by Clang, using the C++ rules. This includes:

- Performing template argument deduction and template instantiation.
- Forming implicit conversion sequences.
- Ranking the overload candidates and picking the best viable function.

Then, the selected function is converted into a Carbon function signature, and called using the Carbon rules for performing a function call, including argument conversions.

# Details

## Example

```
```

```
import Cpp inline '''
#include "stdint.h"
void F(...);
template<typename T> void F(T);
template<typename T> void F(T*);
void F(int32_t);
''';

fn CallF() {
  var n: i64;

  // C++ overload resolution rules pick F<int64_t>(int64_t*).
  // This is converted to a synthesized Carbon function:
  //   fn F(p: i64*) -> () {
  //     // call C++ function template specialization
  //   }
  // This is then called using the Carbon rules for function calls.
  F(&n);

  // C++ overload resolution rules pick F(int32_t).
  // This is converted to a synthesized Carbon function:
  //   fn F(n: i32) -> () {
  //     // call C++ function
  //   }
  // The call is then rejected in Carbon because i64 can't be converted to
i32.
  F(n);
}
```

## Viability versus validity of making a call

These rules mean that we make a distinction between determining whether a candidate from C++ is *viable* – determined using the C++ rules for implicit conversion sequences – and whether that candidate is actually *callable* – determined using the Carbon rules for implicit conversion. Notably, this is the same process that happens in C++ already, except that the callability check is performed using the C++ rules for implicit conversion instead of the Carbon rules.

For this to work well, we need implicit conversions in Carbon to line up reasonably well with implicit conversion sequences in C++, but that is a constraint that we would want to impose regardless. There are two ways we can see a divergence:

- It can happen that a C++ implicit conversion sequence can be formed but a Carbon implicit conversion cannot be performed. For example, for a parameter of type `int32_t`, the C++ logic may decide that an argument of type `int64_t` is acceptable but the Carbon implicit conversion rules would not permit the conversion. In this case we will

select the C++ candidate taking `int32_t` anyway, and reject the call. This is analogous to cases in C++ where overload resolution selects a function that it believes was the intended callee because an implicit conversion sequence could be formed, but rejects the call because the conversion is not actually possible.
- It can happen that a C++ implicit conversion sequence cannot be formed but a Carbon implicit conversion can be. This should be rare, as it is in C++: Carbon is generally more restrictive about which implicit conversions it permits than C++ is. However, there may be implicit conversions that are defined in Carbon code but that cannot be represented in C++, and so the C++ search for an implicit conversion may not find them. When this happens, either the call will be rejected because there are no viable candidates, or the call will select a different candidate that doesn't rely on the problematic Carbon conversion. The latter case is potentially a concern, but conversions that cannot be lifted from Carbon to C++ should be rare.

### Non-overloaded functions

When a call is made from Carbon to a C++ function that is *not* overloaded, these rules reduce to two steps:

- Ensure that the C++ rules believe that the function is viable for the call.
- Call the function using the Carbon rules, as if it were a normal Carbon function.

It is tempting to remove the first step in this case, so that the rules for a Carbon -> C++ call of a non-overloaded function are exactly the same as the rules for a Carbon -> Carbon call of a non-overloaded function. However, doing so would make the interop behavior less consistent. The extra check also doesn't harm our migration story – migrating the callee from C++ to Carbon removes the extra check, but that doesn't affect any callers that were previously valid.

## Future work

### Implementation of the `Call` interface

A C++ overload set should eventually be modeled as providing a templated impl of the `Call` interface, with a predicate constraint that checks whether the function is callable from C++. In approximate Carbon code, this might look like the following:

```carbon
// Built-in mechanism to call into C++ overload resolution, provided by C++
// interop.
fn! SelectCallee(F:! Cpp.OverloadSet, ...template each T:! type)
    -> Optional(Cpp.Candidate);
predicate IsCallable(F:! Cpp.OverloadSet, ...template each T:! type)
    = SelectCallee(F, ...each T).HasValue();

impl forall [...template each T:! type] Cpp.F as Call(... each T)
    if IsCallable(Cpp.F, ... each T) {
```

```
  fn Call(...each t: each T) -> auto {
    SelectCallee(Cpp.F, ...each T)(...each t);
  }
}
```

Any other information made available to the `Call` interface, such as the expression category and constant value of the arguments, and whether the arguments are tuple / struct literals, should also be made available to C++ overload resolution.

# Alternatives considered

## Use the Carbon rules alone

### For template argument deduction

Calls will be made from Carbon to C++ overload sets that include function templates. When this happens, we could either mirror the template into a Carbon template, or we could treat the template as a C++ template and use the C++ rules for argument deduction and substitution.

In general, producing a Carbon template that exactly matches a C++ template is a very hard problem, because the exact semantics of C++ is exposed in its template machinery, for example through SFINAE, and through cases where multiple language semantics are modeled with the same C++ syntax.

For example, an expression such as `T(args...)` in an expression in a C++ function template could be a constructor call, or could call a conversion function, or could be interpreted as a C-style cast if the size of `args` is 1, or could be a no-op if `args` is already a prvalue of type `T`, or could be value initialization if the size of `args` is 0. If the chosen interpretation is not valid in certain ways, the enclosing function is not a candidate for the call, and if it's not valid in other ways, the program is ill-formed. The order in which those checks for invalidity are performed is subtle and programs rely on it. Producing a Carbon template that is identical in all of those details would be extremely difficult, even if we had sufficient language constructs to represent all of the details, which we currently do not.

Therefore the only feasible option appears to be to perform the work of function template argument deduction and substitution using the C++ rules.

### For viability of overload candidates

While we don't yet have full rules for overloading in Carbon, we can at least determine what the outcome of overload resolution would be for some simple cases where there is only one viable candidate. For example:

```
class C {}
```

```
impl i64 as ImplicitAs(C);

// Placeholder syntax
fn F overloaded {
  fn (n: i32);
  fn (c: C);
}

fn CallF(n: i64) {
  // Calls C overload - other overload is not viable.
  F(n);
}
```

But this gives a different answer than we would get for the equivalent code in C++:

```
// cpp.h
class C { C(int64_t); };

void F(int32_t);
void F(C);

void CallF(int64_t n) {
  // OK, calls int overload, truncates n.
  F(n);
}
```

Therefore if we use the Carbon rules for overload resolution, we do not preserve the behavior of the C++ API when used from Carbon:

```
// carbon
import Cpp library "cpp.h";
fn CallF(n: i64) {
  // Calls C overload.
  F(n);
}
```

In order to preserve the behavior of C++ interfaces when used from Carbon, we should not give a use of that interface a different valid meaning than the meaning that it had in C++.

### For ranking overload candidates

After we have determined a set of viable C++ candidate functions, we could use Carbon rules to determine which function should be selected. We don't yet know exactly what those rules will be, but based on our approach for `impl` selection in general, it is likely that candidates will be ranked by some explicit mechanism, such as a prioritized list, at least in cases where a best match is otherwise non-obvious.

However, we reject this alternative:

- Such a ranking does not exist in C++ code, so performing a Carbon-like ordering would likely involve making an arbitrary choice of candidate
- Selecting a different candidate than C++ would select among the same set of viable candidates would give surprising outcomes that may cause the C++ code to behave in unexpected ways

## Use the C++ rules alone

Given that we decide to use the C++ rules to select an overload from a C++ overload set, we could use the C++ rules to build the call as well. This would mean that we:

- Marshal the call argument expressions from Carbon into C++
- Ask the C++ compiler to form a call
- Marshal the result expression from C++ into Carbon

This superficially sounds like it would provide unsurprising results: uses of a C++ API would have the C++ meaning. It would also allow what seems like a clean implementation strategy, where the only interaction between the C++ and Carbon toolchains for a call would be to pass semi-opaque expressions from one language to the other. However, it gives undesirable outcomes for simple cases, such as the following:

```
// cpp.h
void F(int32_t n);
void CallF(int64_t n) {
  // OK, truncates n.
  F(n);
}
```

```
// carbon
import Cpp library "cpp.h";
fn CallF(n: i64) {
  // OK?
  F(n);
}
```

```

The Carbon call to `F` should not be valid: Carbon code should not be able to convert from `i64` to `i32` implicitly. But it is valid in C++, so if we use the C++ rules to build the call, including argument conversions, then this code would be accepted.

It is important that simple C++ functions, such as the non-overloaded `F` function above, have essentially the same call behavior when called from Carbon as corresponding Carbon functions. Therefore the argument conversions for the call should be performed using the Carbon rules.
```