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

Weekly Carbon C++ Interop minutes (rolling)

PLEASE DO NOT SHARE OUTSIDE CARBON FORUMS

Sep 2, 2025

Attendees:

- [ivanaivanovska]
    - Please prioritize reviewing PR #5891 (Support for C++ Overloaded Functions) (zygoloid is currently set as a reviewer)
      The PR already:
        - changes the call to a single function (same rules apply as for any overloaded function)
        - enables:
            - support for overloaded functions
            - support for overloaded constructors (at least call of a single constructor)
            - (WIP) support for overloaded methods (at least call of a single method)
        - type mapping:
            - Carbon -> C++ mapping of **all** builtin types (bool, char, iN/uN, fN)
            - Carbon -> C++ mapping of Record Types (classes, structs, enums)
            - (WIP) Carbon -> C++ mapping of Enums
            - (WIP) Carbon -> C++ mapping of String -> std::string_view
            - Pointers, Const
        - thunks

Aug 26, 2025

Attendees: bricknerb, chandlerc, ivanaivanovska, ragh, zygoloid

- [bricknerb]
    - Started looking into `char`. Draft: #5988

        - C++ Primitive Types issue (#5263) only includes `signed char` and `unsigned char`. It assumes we want them equivalent to `i8` and `u8` (and different from `char`).
            - Do we want to keep them separate?
            - Are we concerned about mapping them to the same type?
            - Should they be `Cpp.signed_char` and `Cpp.unsigned_char` instead (similar to `Cpp.long`)?
        - What about `char8_t`?
        - What about `char16_t` and `char32_t`?
        - Decision: signed char and unsigned char should map to i8 and u8. char to char. Rest should be TODOs.
    - Demo:
        - Can we merge the examples so we can collaboratively work on them?
    - Next features:
        - **Overload resolution for constructors**
        - Features for string support: char array (null terminated?), std::string_view, std::string
        - nullable pointers
        - vtable
        - **operators (with overload resolution)**
            - **+ (not for strings), -, comparisons, shift left**
        - long, long long
- [ivanaivanovska]
    - #5891 review (overload resolution)
- [Ragh on behalf of Richard] Next features.
    - geoffromer is working on optional, which we need for nullable pointers.
    - dwblaikie is working on vtables.
    - chandlerc is working on C++ standard library support.
        - Adding support for includes is easy.
        - For linkage we build on demand during linkage, which is slow, so a lot of the work is adding a robust cache, learning from the experience with Clang.

Aug 12, 2025

Attendees: bricknerb, jonmeow, zygoloid

- [bricknerb] C++ thunks
    - Working on return values (will try to put out a draft PR).
- [zygoloid] Working on a document of demo for milestones.
    - Demo will use RE2, focusing on P0, then P1 and then P2.
        - This will need string support, character types were added last week.
- [bricknerb] Ivana's overloading resolutions PR needs some attention as she was hoping to get feedback before she's back..

Aug 5, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow, ragh

- [ivanaivanovska]
    - Overloaded functions access (visibility): name lookup vs call
        - [jonmeow] After name lookup, form a new instruction that contains the visibility information. After overload resolution, use this information to do an extra check.
        - [bricknerb] This can wait for a followup PR.
        - [jonmeow] Yes.
    - [Carbon/C++ interop] Add support for C++ overloaded functions #5891 - open for review
- [bricknerb] C++ thunks
    - What mechanism should we have to dump the AST for tests?
    - [jonmeow] dump_ast flag to propagate and dump similar to other dump flags. dump_stream adds a stream.
        - Add dump_cpp_ast similar to dump_sem_ir (compile_subcommand)
        - dump_stream on check options, add dump_cpp_ast_stream
        - In check, if dump_cpp_ast_stream is present, dump the ast
        - note, assuming we want full ast, not just thunks, but check with zygoloid

Jul 29, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow, zygoloid

- Overloaded functions:
    - Deducing the type of initializer lists passed as call args (e.g. `Cpp.foo({})`)
    - jonmeow: For initializer list, should be a tuple literal ( )
    - We could defer this case and not support it for now, regardless of the number of overloads.
    - zygoloid: For struct literals, treat as named initializers in C++ (when directly passed)

None

```
// Reject
var x: auto = {.a = 0};
Cpp.Foo(x);

// Make work; different expression category, special-case
Cpp.Foo({.a = 0});

// Note casts remove all ambiguity
Cpp.Foo({.a = 0} as Cpp.FooArgT);

// Reject; Initializing category
fn F() -> {.a: i32};
Cpp.Foo(F());
```

- [bricknerb] Thunks: How to check if a parameter is int32 or int64. Discord. PR.

    - zygoloid: We could add these types when we map the builtin types, so we know these are complete.

Jul 22, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow

- Summit followup? No.
- zygoloid's change (more primitive types support): No one else is currently working on that.
- Overloading: Change is relatively big. Currently pursuing change the logic always: Always use overload resolution. Some issues are left but trying to do mapping for all types, and not do it gradually as it's close to done.
- Brainstorming - we need to talk to chandlerc on whether and how to proceed.

Jul 8, 2025

Attendees: bricknerb, chandlerc, ivanaivanovska, jonmeow, zygoloid

- [jonmeow] Summit discussion
- Thunks
    - Look at the `&` operator logic.
    - Later: We'll have to think how to implement parameters by r-value, but for now we can do copy/move.
    - Return value might emit an emplacement new expression on the thunk side. As if `new (return_address) auto(C++ call)` using `operator new(size_t, void*)` in `<new>`.

Jul 1, 2025

Attendees: bricknerb, chandlerc, ivanaivanovska, jonmeow, zygoloid

- [bricknerb] Calling thunks.
    - Callee function: It seems like we need the `llvm::Function` for the thunk in order to call it. Currently we generate `llvm::Function` from `SemIR::FunctionId`. Do we want:
        - The thunk function to have `SemIR::Function` like information (with or without `SemIR::FunctionId`) with modified param types so we can use that to generate the `llvm::Function`?
        - Generate the `llvm::Function` based on the original `SemIR::Function` fields by doing the pointer types generation in Lower?
        - Use Clang on the C++ thunk function to generate `llvm::Function`?
    - Args: Should we create the pointers (take the addresses of the original args) in Lower or in Check?
        - If in Check
            - Should it be part of the extra thunk information in SemIR?
            - Should we create dedicated instructions?
        - If in Lower
            - How do we take the address of a value?

        - [chandlerc] Idea is to only have specific types (i32, i64, pointers) in the signatures of the functions that appear on the boundary. If the function doesn't have such a signature, Carbon generates a thunk declaration with a simple signature, Clang generates a thunk definition.
            - [jonmeow] When making a call to a function that has a simple signature, use Clang to mangle.
            - [chandlerc] Patch Clang to create the thunk – create AST nodes.
            - [bricknerb] Creating a Clang FunctionDecl.
            - [chandlerc] Can create an extern "C" function in order to disable mangling.
            - [zygoloid] Better: use AsmLabelAttr to fully control the mangling.
            - [chandlerc, jonmeow] Check should create a declaration of the thunk so that we can build and type-check calls to it in SemIR.
            - Use "<C++ mangled name>.<some suffix>" as the mangled name for the thunk. This will then demangle following the C++ mangling rules.
                - Reserved by itanium ABI
                - ".carbon_thunk" as a suggested suffix
- [bricknerb] Probably premature: How will thunks handle virtual inheritance?
    - [chandlerc] If we don't have an exact match for pointer type, don't use the simple ABI and let Clang do the pointer conversion.
    - [bricknerb] This means that the thunk generated depends on the argument type.
    - [zygoloid] Suggest performing the conversion on the Carbon side, or deciding we don't want to support these implicit pointer conversions in Carbon.
    - Perhaps we support the "hard" C++ conversions by generating another kind of thunk where the C++ implementation only performs the conversion and returns the converted value. Then can have only one thunk per function.
    - Need to decide how to handle multiple inheritance in general. Eg, which C++ base class is "the" base class according to Carbon. Unclear which pointer conversions should be handled by a conversion thunk versus handled directly.
    - Conclusion: don't worry about per-call thunks for now at least.

Jun 24, 2025

Attendees: bricknerb, chandlerc, jonmeow, zygoloid

- [bricknerb] Started looking into Thunks and still need some more time to start something.
- [zygoloid] Primitives proposal in good shape.
    - [chandlerc] Have some time to look into that.
- [chandlerc] Ref proposals going to get unblocked this week and perhaps land. References in interop might be something we can start looking into, though it's just the design.
    - Pointers are probably better to start earlier, though it's just non-null for now. Null pointers are being designed.
        - `this` pointer is not null.
- [jonmeow] #define integer literal constants is something we can look into now, and is not waiting for anything.
    - [chandlerc] We should look into how swift is doing that and clarify if we do something different.
    - [jonmeow] Stuff we could test: INT_MAX INT_MIN…

- [jonmeow] Carbon needs to design string literals so we can do system apis.
    - [chandlerc] We know what we want to do but someone needs to write it down, and then we also need to implement it.
- [chandlerc] Can we do proposals that are brain dumps that are not full designs (which would require research)
    - [jonmeow] We can use design ideas in GitHub, and we have used it in the past.
    - [chandlerc] Maybe I'll try that.

Jun 17, 2025

Attendees: ilya-biryukov, ivanaivanovska, jonmeow, zygoloid

- [ivanaivanovska]
    - Overloads: Discuss how to design a new instruction for representing the overloaded function set.
    - Planning to add a new instruction that would propagate the results of name lookup until the call.
    - Not sure if there were already some discussions on how we are going to handle the results of the overloading from Carbon or only C++ functions.
        - [zygoloid] For Carbon we will be dealing with functions completely differently, let's do them independently.
    - [zygoloid] if we can represent it on the Clang side as `OverloadExpr`, we can have only a single thing. However, storing an expression in Carbon is not something that is easy and there isn't even a DeclID equivalent for expressions.
    - [zygoloid] so maybe we should have a list of function declarations on the Carbons side?
    - [jonmeow] do we want to support Carbon overloads mixed with C++ overloads, e.g. let's say you are inheriting from the C++ class and you want to provide overloads with the same function name.
    - [zygoloid] the way two overloadings work is sufficiently different, we will need to figure out how to design it first.
    - [jonmeow] how do we want to show the overloading errors?
    - [ilya-biryukov] I propose to just show the C++ errors, at least while the overload set is purely from C++ functions
    - [jonmeow] the errors would still be useful.
    - [zygoloid] in terms of source locations, we can provide something that points back to Carbon.
    - [zygoloid] we don't want to have pointers from SemIR to Clang AST. A thing with persistent handle is a decl, so the options are either:
        - **A list of Clang declarations in whatever format we're using.**
        - We somehow wrap `OverloadExpr` on the Clang side. But that seems hard.
    - [jonmeow] that accounts to creating something similar to function-id and function
    - Which type does it have?
        - [zygoloid] maybe a special singleton type.
        - …

    - [jonmeow] in Carbon when we create a function it has its instance of function type and the type of function type is type. **The new instruction should probably be a type instruction similar to function type.** And so that you can treat that value that's returned by member access as a subvalue.
    - Note: this probably doesn't work today if you write this in a test.

None

```
let x:! auto = Cpp.Overload;
x(...);

fn CallOverload[template T:! type](x:! T) {
  x(1);
}
```

    - [ilya-biryukov] Carbon overloads efficiency compared to C++
        - [zygoloid] Closed overload sets means caching will work
        - [zygoloid] Ranking of candidates can throw out a lot based on types in signatures based on conversion approaches
        - [zygoloid] Because integer type is a parameterized type, having one type covering a handful of overloads. But things with size-based choices work differently.
- [ivanaivanovska] Primitive types proposal: clarify naming for floating-point types:
    - C++->Carbon: float/double -> Cpp.float/Cpp.double?
        - Are they going to be just aliases to f32 and f64?
        - [zygoloid] yes, we should have those types even if there's fixed mapping to Carbon.
    - Carbon->C++: `f32`/`f64` -> `float`/`double`?
        - Then it wouldn't be bidirectional as float in C++ will have a type Cpp.float in Carbon, which will be type-aliased to f32. The other way around it will be the same, but is it okay that it skips the `Cpp.float` type?
        - …
        - [jonmeow] or we can have distinct Cpp.* types and define implicit casts.
        - Having the same type for as many cases as we can should lead to better ergonomics, it's definitely a trade off.
- [ilya-biryukov] Updated patch with in-flight Clang thing. Still in draft, but working as intended. Seems to be working well. Produces LLVM module that we later link. Can probably get rid of CPP code generator in Carbon completely, in another patch.
    - http://github.com/carbon-language/carbon-lang/pull/5543
- [zygoloid] what's the best thing that I can be working on for interop, will have some time soon?
    - [ilya-biryukov] roadmap and longer-term plans.
    - [jonmeow] choices and enums and what kind of mapping we would need between the two. More generally, what we don't have the mapping for yet.
- [zygoloid] Carbon/C++ interop is now usable from Compiler Explorer: https://godbolt.org/z/8djKqaM6a

Jun 10, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow, zygoloid

- [jonmeow] Two different carbon files including different sets of headers could have the same type and have different decl ids and we will need to reconcile that. Long term.
    - Note, this is file A includes file B and C, both of which expose the same C++ type on API boundaries. File A needs to see the type as the same.
- [ivanaivanovska] Overloaded Functions
    - Started looking into implementing support for overloaded functions. Looking into creating OverloadedCandidateSet and calling BestViableFunction to get the Clang overload result. Any inputs that may help would be highly appreciated.
    - [zygoloid] Mapping of Carbon types to C++ types (reverse of what we have)
    - [zygoloid] Also Carbon Expr to C++ value kind (references to lvalue kind)
    - [zygoloid] Mapping of Clang types to Carbon types. OpaqueValueExpr: Placeholder expression that Clang supports and use it as the argument expressions of the function.
    - [bricknerb] Regarding thunks question, will be somewhat independent, will determine the overload then generate the thunk and call it.
        - [zygoloid] Agreed can keep it separate
    - [zygoloid] We should choose the overload before generating the Thunk - one thunk per C++ function with specific argument types. Conversions are done on the Carbon side.

Jun 3, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow, zygoloid

- [ilya-biryukov] Sorry, cannot attend. I am working on finalizing a PR to interface with Clang APIs more closely. It's coming together, just need more time to flush out the details, hope to send for proper review tomorrow.
    - http://github.com/carbon-language/carbon-lang/pull/5543
- [bricknerb] How to learn about SemIR in a more formal way?
    - [jonmeow] The recent issue was something that was actively worked on.
    - [jonmeow] A recorded talk might not give more information.
    - [jonmeow] Call is extra complex than regular SemIR.
    - [zygoloid] Maybe we should have SemIR reference to explain what it means.
    - [zygoloid] LLVM has a verifier that helps with checking of the right values. We could do something similar.
    - [ivanaivanovska] External documentation could help since right now we have to look at PRs to understand what was the idea. The documentation we have is useful but it has low coverage.
    - [jonmeow] We could have better documentation in the typed inst. It's somewhat similar to LLVM IR but it might not be useful to learn more about it because it's also significantly different.
    - [zygoloid] It's not obvious what documentation is useful, so pointing out what documentation is missing and where we looked for it would be good to know.
- [ivanaivanovska] Primitive types proposal. Kate is the lead reviewer, but I got no feedback, how should we proceed?
    - [zygoloid] I'll make sure it will be looked at.
    - [jonmeow] It shouldn't block progress.

May 27, 2025

Attendees: bricknerb, ilya-biryukov, ivanaivanovska, jonmeow, zygoloid
Topics to discuss:

- [bricknerb] confirming the direction with “simple ABI” functions. Currently thinking about passing a `struct` by value as the first example.
    - [zygoloid]: two other options:
        - Build CallExpr that we later turn into LLVM IR during lowering
            - Representing the arguments is likely to add a lot of extra stuff.
            - May generate less AST on the Clang side, probably not a priority at least until 1.0 release.
        - Building a complete function decl on the Clang side for the wrapper that we can call on lower.
            - Simpler.
            - Probably the way to go.
            - Costs more on the AST
    - [zygoloid]: on the Carbon side, the problem is much simpler because we control the calling convention. Still a question of whether we do something uniform like passing everything as a pointer or passing some things by value.
    - [ilya-biryukov] For types that are not copyable and not movable, C++ allows us to eliminate copies and we cannot emulate that over the AST reflecting those semantics by just passing pointers.
    - [bricknerb] we could also avoid passing pointers for primitive types.
    - [jonmeow] I would maybe propose to start with pointers for everything, maybe not even worth special-casing primitive types and optimizer can take care of that.
- [ivanaivanovska] Where, in Carbon, should we define C++ types like `long`.
    - `Cpp.long`
    - `Core.Cpp.long`
    - A different library?
    - Should we require `import Cpp` to use them? This seems limiting.
    - [zygoloid] thinking ahead about Carbon<->Rust interop that we will have some day, there should probably be `Cpp` namespace.
    - [jonmeow] so it’s `Cpp.long` rather than `Core.Cpp.long`
    - [bricknerb] can one use `Cpp.long` even if you don’t import any C++ headers?
    - [jonmeow] we could allow `import Cpp;`
        - That matches our default library import syntax.
        - Implementation-wise gets into a different quirk. Might be a special-cased library for C++ (written in Carbon code and it’s the one file that is allowed to declare itself as `package Cpp`).
        - The argument for special-casing it: technically it’s C++ types mapped to Carbon types. You can treat them similarly to `std::string` → Carbon string.
    - [zygoloid] `Cpp.long` is a compelling name, following the pattern of `Cpp.whatever` gives you `whatever` from C++. However, `long` is the only type where this works. Doesn’t work for `long long, unsigned long, …`
    - [zygoloid] That might be an argument for `Core.Cpp` rather than just `Cpp`.
    - [jonmeow] Chandler was relatively comfortable about the idea of `long_long`. If people defined it in C++, it wouldn’t work for them. The conflicts seem unlikely.

    - [zygoloid] Could maybe consider other "fun" options like `Cpp.unsigned(Cpp.long)`, but probably not a good model.
    - [ilya-biryukov] `long_long` as an identifier does appear to have significant usage in C++.
        - Not sure how significant, definitely used in a few places (python runtime is worrying, although it seems to be only in one `.c` file).
    - Other options: `Cpp.std.long`, or `Cpp.long_long` can be shadowed by anything user-defined.
- [ilya-biryukov] looking for early feedback on one approach for a more tight Clang integration (replacing ASTUnit with a more tightly controlled alternative). It isn't ready yet, but how do folks feel about bridging the callback API and Carbon's API via std::thread.
    - Is this going to fly or should we look for alternatives?
    - https://github.com/carbon-language/carbon-lang/pull/5543

May 20, 2025

Attendees: bricknerb, ivanaivanovska, jonmeow, zygoloid

- [zygoloid] inline PR: propagating parameters for Clang to generate the AST. Passing a factory function to generate the AST could be better, but let's wait with that for when we use modules (pcms).
- [jonmeow] Use decl ids instead of pointers. decl ids are not created before serialization. We could serialize and deserialize. We want it now to avoid dependency on pointers.
    - [zygoloid] We could have a table instead of serialize and deserialize.
    - When check is finished we want an AST (pcm), so we'll have it when we import the Carbon library.
    - Short term: We could #include the files for a specific Carbon library.
    - We could have a tag that is either an id or a pointer.
    - From space perspective it would be better to have a side table.
    - Clang has a `LazyDeclPtr` that converts declid to pointers.
    - Not just functions.
    - Conclusion: Side table containing pointers. Long term it would contain the lazy decl ptr. Side table would need to be serialized when we serialize a Carbon library.
- [bricknerb] Primitive types: we're learning from how Rust is mapping these.
    - We would go with a few types for now, and keep the rest open for later.
    - [ivanaivanovska] Do we want aliases in the Carbon side that depend on the platform. Eg for int_fast32_t, int_least32_t
        - [zygoloid] Let's not include them for now and wait until we need them.
    - [ivanaivanovska] Do we need both the DD and the proposal?
        - [zygoloid] Eventually we want the proposals merged into the DD.

May 13, 2025

Attendees: bricknerb, chandlerc, ilya-biryukov, ivanaivanovska, jonmeow, zygoloid

- [ilya-biryukov] alternative to interfacing with clang through AST Unit: freezing Clang in the middle of "compilation" and reusing the whole pipeline.
    - API not available yet, but in the works.

    - Richard: Sounds like a reasonable approach.
    - Should we still pursue ASTConsumer in AST Unit in the meantime (or alternatively)? Let's pause for now.
- [jonmeow] inline warnings vs errors
    - For calls to forward declarations (`inline void foo();`)
    - Pushing Carbon for correctness because this can be a miscompile source
    - Versus migration costs and requiring C++ users to clean up prior to migration
    - [chandlerc] Do kind of want to push users to clean up code on migration. If there are things to make interop experience significantly better by pushing a little change, that's okay. For example, forcing users to compile with Clang (some C++ not compatible)
        - Some places (not here) we can have the quality of interop increase based on changes to the code.
        - Unless there are particular examples where this is violated and would be a barrier to migration, would be inclined to support stricter model. Would similarly be inclined to enable sanitizer errors where C++ doesn't.
        - A related example is null "this" pointers. Bad C++, compilers allowed it, code abused it; probably don't want to facilitate it, instead force users to update C++ code.
    - [ilya] Is this really a problem at all for code? Expect null "this" is a bigger problem. Feels like we can be as strict as we can, adjust if there are problems. Always easier to loosen rules later.
        - Would be nice to have a framework for deciding these, when disagreements pop up.
        - Also would be nice to have good ways to do this. Upgrading warnings to errors is probably easy. Probably not worth doing when there's a big patch to Clang required.
    - [chandlerc] inline function could be a bigger problem because they're likely to occur in headers. Did a lot of things in Clang to work around standard library headers. If it's in some standard Boost library, maybe we can't avoid it.
        - High level strategy would be good to write down.
        - Generally want C++ code to be modern and clean, probably generally want -Wall to be clean. Fine to back off, but where we back off, there should be motivating examples (documented).
        - With C++ code we compile, target high-quality C++ compilation story. Modern Clang, modern warnings -- the things we'd typically recommend to people.
        - Second tier of C++ code which we compile, but don't interop with. Hold a strong ABI story for accessing vanilla C++ system ABI, but be faithful and make sure it's a good story for precompiled artifacts. Maybe have to tag everything that crosses that boundary. Maybe can't use standard library types because we may customize STL types. Just need to make sure there's some bridge across that boundary. Everything about the level of C ABI is "nice-to-have".
    - [jonmeow] When talking about this high-level strategy, it makes me think myabe we need a more formal version of the "building code" vs. "HOA-rule" to apply back here.
        - Would let us apply back to guide on what should be a warning vs. an error

        - Maybe useful to apply generally to Carbon errors vs lint messages
    - [bricknerb] Is the approach to be very strict and when it becomes an issue, remove that strictness; or, allow users to configure through flags instead of having one version of diagnostics applied
    - [zygoloid] Some sympathy for wanting the stricter mode, but also worried about the impact on folks starting to use Carbon within an existing C++ codebase
        - Adding impedance there might turn people away
        - For the original inline forward declaration case, it's actually ill-formed C++, could maybe make Clang stricter?
    - [chandlerc] Regarding impedance, should see how much it is and whether it causes people to struggle to try out Carbon. But, can probably start with a high quality bar.
        - Probably can't make people make changes that aren't based on C++. Start with FIRST principles and point out things which are bad code, irrespective of Carbon. Easy to make the point to fix up front.
        - Other part, two things regarding HOA vs building code thing.
            - Would like it if we have basically no warnings in Carbon. Should only have things that are more just things that aren't really compiler positions, letting people choose coding conventions.
            - For Clang warnings that are sufficiently strong quality, kind of treat them like errors; for other things, maybe think of them as coding conventions or linter errors.
        - On flag configuration, don't think we can come up with a policy up front. Need a particular example to motivate it, decide whether it should be a configuration thing or not. See the concrete cases that motivate it.
    - [bricknerb] Once Carbon is more mature / being used, will we learn about cases that cause problems? Or will we not care given it is mature?
        - [chandlerc] (from Discord) To an extent, I'm less worried about that -- as the project gets more mature, if anything folks should be more and more motivated to let us know if there are barriers to adopt that we need to work on
          i would think its in the early phases that we need to be careful about not hearing folks hitting issues
          but I think we can do stuff there such as actively reaching out to folks who are adopting and creating explicit avenues to get feedback, etc

May 6, 2025

Attendees: bricknerb, chandlerc, ilya-biryukov, ivanaivanovska, jonmeow, zygoloid

- [ilya-biryukov,iivanovska] What is the process for document approvals? When should we consider a document done?
    - Need to get the lead's approval.
    - More formally: write up a proposal and go through a language approval process.
    - Less formal: file a lead's question and ask how to proceed.
    - Covered in evolution.md file.
- Lead's question: are we okay with the current state of the document on primitive integer types?

    - Richard: it looked fine last week haven't looked since then.
- [ ] AI: start the formal approval process and start the formal review process of the proposal for mapping primitive numeric types.
- [jonmeow] should C++ reference translate to Carbon pointers or should we wait for the new reference design first?
    - [chandlerc, zygoloid] references in parameters are likely to change, reference in struct fields should translate to pointers.
    - Reference return types may turn into pointers: seems okay.
    - Reference on variables: should probably translate into pointers, not the highest priority thing to worry about.
    - Final thing: reference types used in template arguments. Roundtripping closely seems important there, maybe we'll end up with C++ reference wrapper type there.
    - Clarification: to const pointers to avoid reassigning if that's required.
    - We can possibly distill this to a simple rule. When type identity is not essential to language semantics, we expect references to map to a Carbon type (either pointer or a reference binding that are about to show up in the next month). When it's significant, we want to build a Carbon smart pointer that precisely models C++ references.
- Insights about pointers.
    - We can leverage annotations on pointers that Clang provides. We want Carbon pointers to not be nullable and use optional pointers instead. `_Nonnull` C++ pointers map directly to Carbon pointers (there's a file-level annotation to make this a default).
    - For Carbon-to-C++ (bidirectional mapping): for API boundaries, we probably want to map Carbon pointers to _Nonnull C++ pointers (and vice versa).
    - Caution: the _Nonnull annotations are not part of the type. Once we hit templates and type identity, things get tricky. We might end up needing different mapping where type identity matters.
- Do we need to wait for pointers for Thunks?
    - We can start with guaranteed non-null pointers and have a simple correspondence.
    - …
- [bricknerb] Does it make sense to manually add `used` attribute to each inline function to trigger code generation for the Clang module before linking it to the Carbon module or am I missing something here?
    - Context: https://github.com/carbon-language/carbon-lang/pull/5427
    - Approaches tried so far:
        - tried a single llvm::Module, but this only works with internal Clang APIs.
        - with two independent llvm::Modules linked together, couldn't get Clang to generate the IR for inline functions.
    - Trying to use two modules seems okay, a single llvm::Module seems harder.
    - In the long-term we want to call into sema to mock a function reference to a function. In addition to setting the used bit, it'll also trigger necessary template instantiations and produce any warnings that only happen at that point.
    - The most probable reason it isn't working: expression evaluation context in the default state is that we're in some top-level unevaluated context. We're likely missing a push to the evaluation context (there are RAII objects for that).

    - In the short-tem to unblock progress, used attribute might also be okay.
- [chandlerc] It would be helpful to invest into debugging tools.
    - Concretely, adding flags to dump LLVM IR (at each step, e.g. for each of the two modules and for the merged module)
    - Dial up the verbosity in Discord: “live blog” the progress, even on small steps. We can create busy channels, it’s okay to overcompensate in that direction.
    - What are better times of day / better days of the week for the Munich folks? Please share offline.

Apr 29, 2025

Attendees: jonmeow, zygoloid, ivanaivanovska, bricknerb

- [jonmeow] Diagnostic format, importance of putting "in import" first
    - Context: https://github.com/carbon-language/carbon-lang/pull/5246#issuecomment-2784301206
    - In C++ we see `in file included` lines.
    - Currently the location is added as a note.
- [bricknerb] How to synthesize code for interop?
    - https://discord.com/channels/655572317891461132/768530752592805919/1364942078207070240
    - Overloading is separate from synthesizing. When we have a single function after overload resolution and then the call from Carbon will use synthesized code.
    - Step 1: Do the lookup and return a set of overloads to Carbon (happens on name lookup in Carbon). Carry around something like OverloadExpr in Carbon?
    - Step 2: Do the overload resolution and template argument deduction to pick a single concrete function that we need to call (happens on function calls in Carbon).
    - Step 3: Lower the call to LLVM through a wrapper function in Clang that Carbon calls.
        - The function has a simple ABI to make sure the code generation for it is trivial, e.g. accept all arguments by pointers.
        - On the Clang side, the function should be marked always_inline so it’s optimized well.
        - In lowering to LLVM IR, we can either have one shared or two separate llvm::Module (one for Clang, one for Carbon) that we will link with LLVM.
            - Does a single module have a chance to generate better code?
            - It should not matter if we do the LLVM link step before optimizations.
        - LLVM globals and symbols having internal linkage should not be duplicated, though.
        - Things to worry about: merging of global symbols and symbols with internal linkage. Perhaps look into what Swift does.
        - A single llvm::Module is likely harder to implement, but two llvm::Modules are not very trivial either.
    - Two parallel work threads

        - 1. Step 1-2: implement name lookup and overloading.
        - 2. Step 3: lowering. Also has independent subtasks:
            - Wire up code generation for inline Clang functions (build missing infrastructure unrelated to overloading). It would force us to choose either one or two llvm::Module.
            - Implement the lowering for a single function with arbitrary signatures based on that infrastructure.
            - (late addition): another prerequisite is calling functions with pointer arguments from Carbon.
- [ivanaivanovska] Primitive types.
    - Floating types: Can we map float and double to f32 and f64? Yes. Other types can be ignored for now.
    - We can start with some simple types.
    - External contributor can start. MUC folks can review it first.

Apr 8, 2025

Attendees: jonmeow, zygoloid, ivanaivanovska, bricknerb, chandlerc

- Code review process for MUC (specifically when Jon is out).
    - Be more insistive and make sure someone who is around is assigned.
- Relying on libc++ in Carbon interop tests.
    - Chandler: Figure out how to package libc++.
    - Jon:
        - Short term and in the common case: Small snippets.
        - We should have a few tests that test the real thing.
    - Chandler: Making progress on packaging clang builtins, which is a pre-requisite for packaging libc++. 3rd attempt is looking pretty good.
- Jon: What should `long` do?
    - Chandler: Richard?
    - Jon: long should be i64 for now until we have something better.
    - Ivana: I'm relying on the old design doc.
    - Jon: the old document is mostly correct for now.
    - Chandler: We should have a fresh design doc, based on the old one. The model we want to follow C++ conversions.
    - Richard: We have both mappings to figure out. C++ -> Carbon depends on the platform. Carbon -> C++ should use the std::int* types.
    - Jon: The old proposal had an alternative that we should support so people can write compatible code.
    - Chandler: If we could compile on a 16 bits int, we should support it, but I don't think we'll be able to do that. int64 in C++ should map to i64 and vice versa. long, long long, int should use conversions.
    - Richard: There's always clong and cint etc and we sometimes don't use them?
    - Chandler: We have to support the conditional overloads for the types.
    - Richard: No c-types map to i-n.
    - We need to check offline if int and/or long long map to different types in different platforms.
    - Jon: For now long would be i64?
    - Chandler: If int32 and int are always the same we can do things differently.

    - Chandler: We shouldn't support 16 bits platforms, because it's not worth the complexity for now.
    - Jon: float and double don't have the same concerns?
    - Chandler: I think not and we should make sure that std::float32_t is the same as float.
    - Chandler: We should prototype with int32
    - Jon: We already have that and we should decide about the other primitives.
    - Chandler: floats are ok. float and double are aliases for 32 and 64 bits floats.
    - Chandler: doc about integer and float primitive types.
    - Carbon team will not attend Euro LLVM.

Apr 1, 2025

Attendees: ivanaivanovska, ilya-biryukov, bricknerb

- Boaz: waiting for the review the class change (critical comments addressed)
    - https://github.com/carbon-language/carbon-lang/pull/5156
- Boaz: also started looking at name poisoning, mostly unrelated to interop
- Boaz: made a change piping Clang diagnostics to Carbon ones
    - https://github.com/carbon-language/carbon-lang/pull/5177
- Ivana: working on enabling (primitive integer) parameters for functions. Almost ready to send for review, added tests today. What's left: refactoring the code to share the code between return types and parameters, and a few NITs. I plan to send this for review tomorrow.
- Discussed "Another one: the whole idea of synthesizing some C++ code"...
    - Is it more in the frontend or lowering/middle-end?
        - Ilya: I suspect this would touch a bit of everything in both compilers (Carbon and Clang), Frontend would need to distinguish between Carbon/C++ calls/types and the lowering/middle-end would need to create LLVM IR that uses this custom ABI for the interop boundaries (calls, etc)
    - Should we start on it now or wait a little?
        - We should keep discussing it for now, flush it out a bit more before rushing to implement it. We need to agree on a high level approach, let's talk to Chandler, Richard, Jon and other Carbon folks more.

Mar 25, 2025

- State of the interop document so far
    - Injection is in a better state now
    - Still missing: we should have at least a "parallel file" approach, to avoid carrying patches for the standard library.
        - Use-case: folks adapting Carbon, it'd be nice if they won't need to change code (e.g. if they can't)
        - There are reasons to have either one
    - Another one: the whole idea of synthesizing some C++ code inside of Clang, some Carbon code inside of the Carbon compiler and having a custom ABI for those two pieces to connect rather than having the C++ ABI inside of Carbon and vice versa.

        - Thunks on both sides to have a completely controlled boundary. This allows more flexibility.
        - Extension points for Clang to synthesize AST and on the Carbon side to SemIR.
        - Good way to start: look at function calls and overload sets, see the approach Richard suggested.
        - The idea is to always do this approach, the current approach won't be able to handle overload sets.
        - Non-trivially copyable types are not ready in Carbon yet for something similar.
        - But we can start with trivial types:
            - Overload set that contains a 32 bit integer overload and a float overload. We call it with a 64bit integer on the Carbon side.
            - Carbon picks a float overload, C++ would pick the 32bit integer; but you need conversions that don't exist in Carbon and it would produce an error. But the 8 bit integer **will** be available and the conversion should happen on the Carbon side.
        - It can be prioritized right after primitive parameter types
        - Potentially next: tuple, conversions from multiple tuple element types.
            - Open question: how to homogenize the layout between the C++ tuple and Carbon's tuple?
            - Another open question: how to write an implicit conversion from `std::tuple` to Carbon tuple if Carbon does not have template support?
            - Another potential target for writing implicit conversions.
                - `std::string_view`: easier because it can be just copied
                - `std::string`: harder because copies need heap allocation, etc.
                - `std::tuple`: see the challenges above
    - Types we probably need for the demo
        - string_view
        - span
        - string and vector
- Making the demo more concrete
    - One suggestion is to pick the idiomatic (but very simple) C++ API
    - E.g. something in `util/math` (e.g. `vec2d`)
    - Then a boring function in the string utilities (accepting only string views).
    - Protobuf APIs!

Mar 18, 2025

Notes

- Unresolved questions in the document by Boaz
    - Chandler has more thoughts, but did not get a chance to write them down yet
    - Anything unresolved about compilation model?
        - Correlation of inline Carbon with modules and / or textual include, in particular about proto header may be treated as either textual or modular in different compilations

                - In two different compilations, treating the header differently would be okay.
                - What really matters is that for a given Carbon compilation is either a textual or modular header and treated as such.
                - It may end up awkward. We can diagnose this better if this becomes too confusing for users.
            - Carbon file will map to the C++20 module or a Clang header unit. The Carbon is “inherently” after it.
                - There needs to be a way to find the carbon file corresponding to the module unit.
                - This would help us reduce the need to modify C++ headers.
                - In Bazel cc_library each header is a submodule, so this effectively gives us a Carbon file that we can add per C++ header.
    - Ilya was confused why we require the `#pragma inline carbon` in C++ files if we have wrappers.
        - We want inline Carbon to make it easy for users to add Carbon code.
        - Deferring the pragma is probably reasonable (annoying to implement, needs discussion with Clang), but we want to document this idea and plan for it.
        - They have to be modular headers to make sure the transitive includes work with the “inline Carbon” blocks in C++.
        - Most of the push for pragma is coming from experiences with Go. Creating a separate file is something that developers will do, but a lot of them don’t. Same for SWIG.
- Richard finished document talking about how overloading should probably work
    - Carbon: C++ interop for overloaded functions and function templates
    - Also applies to constructors
    - C++ overloading is not representable in Carbon
    - This has an impact on the migration path, we are going to have more failure modes and we can trade off if that happens at interop time or migration time.

Mar 11, 2025

Attendees: jonmeow, zygoloid, ivanaivanovska, ilya-biryukov, bricknerb, chandlerc

Notes

- Design doc - diagram of multiple toolchain invocations
    - jonmeow: What is the purpose of the bottom diagram?
    - ilya-biryukov: Its purpose is to show how multiple toolchain invocations work together.
    - jonmeow: We need to make a decision on whether the pcm is build output.
    - chandlerc: Why is it essential to output a pcm
    - ilya-biryukov: If you want to import a template instantiation, you need to get a pcm.
    - chandlerc: pcm files are not always a performance win. Might be faster to parse again rather than read all the pcms.
    - ilya-biryukov: #define should not use other imports so we need pcms.

    - zygoloid: We need serialized AST iff we need serialized SemIR. I think we need this sooner. We don't preprocess state to leak between different imports. Even if parsing Carbon is fast, if we need to parse all the transitive imports it will be slow.
    - chandlerc: Not worried about that. Transitive parsing is not scary. We cannot afford designing SemIR serialization this year, even if it's slow.
    - zygoloid: Not talking about prioritization, we need this to make it to scale for google3.
    - ilya-biryukov: The alternative means we'll have multiple ASTs, and then we need to handle merging them. Modules provide the best solution for merging.
    - zygoloid: Separate whether we use pcm files, whether we use Clang's module merging, and whether we use multiple ASTs. We can have a single AST context with multiple ASTs without pcms.
    - jonmeow: You can create the pcms in memory without having them as build output.We might want Carbon serialized state to be in the same file as the pcm.
    - chandlerc: Serialization will cost us 2 quarters minimum.
    - ilya-biryukov: Let's separate the long term solution and the short term solution. Short term means no SemIR serialization this year.
    - jonmeow: Do we want to eventually serialize?
    - chandlerc: Visibility group solution might fall apart because we'll have too much C++ imported. We could have it serialized as Carbon files and pcms. We want it as one file because otherwise it's harder to debug with multiple files.
- Design doc - do we need AST for prelude?
    - jonmeow: We might be able to avoid having the prelude always having AST. If you #include a C++ header. For example, converting std::string with Carbon String might require some magic insertion into the C++ include so we don't have to have an AST for prelude.

Mar 4, 2025 | Carbon C++ Interop

Attendees: jonmeow, zygoloid, ivanaivanovska, ilya-biryukov, bricknerb, chandlerc

Notes

- How to serialize/deserialize the AST decl pointers?
    - File currently points to Compile Subcommand, which we need to change to make it serialized
    - decl id. If we don't have it?
        - Write the PCM before and after
    - Serialized AST would probably help by reducing recompilation of C++ which would likely to be the slow part.
    - chandlerc: We probably don't want to optimize for C++ compilation time and we might not want to change how compilation happens.
    - zygoloid: For API files we want to have it serialized.
    - chandlerc: Agree on API files.
    - ilya-biryukov: Split the discussion. Have a PCM with the Carbon files and the second one might only be needed for performance.

    - Chandler: Why do we need the serialization for the API files? Long term sure, for the short term?
    - zygoloid: If we serialize the SemIR then we also need the serialized AST because they will point to each other. Right now we don't serialize the Sema for anything. I expect prioritization to change when we do more C++ in API files.
    - ilya-biryukov: We can optimize for the pathological cases. The modules work we do might be important for Carbon.
    - jonmeow: Serializing the AST might not require a serialized Sema.
    - zygoloid: Carbon templates which trigger instantiation of C++ template which trigger Carbon templates.
    - ilya-biryukov: We probably want to discuss this over a design doc.
    - chandlerc: We want separate design docs. One just for serialization deserialization. Let's talk about ABI.
    - zygoloid: Whenever we have a call from Carbon to C++, we build a Clang callexpr, so in the long term we want to only create Carbon parts only for Carbon code.
    - Chandler: Create a thunk in the Clang AST and make the call inline in the llvm ir.
    - ilya-biryukov: What is thunk?
    - chandlerc: It's a function wrapper. For example: for calling virtual methods. When a caller and a callee don't match, you typically have a thunk.
        - https://en.wikipedia.org/wiki/Thunk
    - zygoloid: (Folk etymology?) Old machines had a thunk to convert from low power to high power that made a "thunk" sound. Let's start with this and see how it works.
    - chandlerc: I have ideas on how to reduce the overhead. Let's start a design doc and fill in the gaps.

Action items

- [ ]

Feb 25, 2025

- bricknerb: HLD
    - Carbon code -> Compiler -> Take C++ files and generate inline C++ file and a modulemap that bundles everything -> Clang -> Serialized AST -> Compiler (CompileSubcommand).
    - jonmeow: Generate C++ after parse, after serialized AST can go to check
    - zygoloid: One Module per Carbon file/library.
    - ilya-biryukov: Ivana is already working on serialization/deserialization.
    - zygoloid: Serialization might be useful when Carbon imports in the api.
    - jonmeow: We're not blocking on serialization, we have it in memory.
    - zygoloid: Clang `CodeGen` only emits inline / template things that are referenced / needed. And when importing a module we eagerly emit things that need to be emitted with every use. So doesn't matter which one we use for generating the right IR.

    - jonmeow: From build perspective, building the modules could be cached and the C++ is expected to be the slow part.
- bricknerb: Using AST.
    - CompileSubcommand owns the AST and it's populated when handling Cpp imports.
    - NameScope will optionally (only used for Cpp namespace) have a lazy import behavior that it can trigger when it fails to find a name.
        - The Lazy import would search the AST for the name, and if found, populate the Context with the information.
        - This means the Lazy import would contain pointers to the Context and the AST.
        - The lazy behavior means that NameScope might not have a "const" Lookup behavior.
    - jonmeow: Instead of NameScope, look into name lookup for import IRs (cross-package)
        - Around `ImportNameFromOtherPackage`
    - zygoloid: We want to be consistent with C++ around flags.