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

Open discussions minutes (2021 Jan-Aug archive)

PLEASE DO NOT SHARE OUTSIDE CARBON FORUMS

2021-08-12 part 2 indented to the left

- Attendees: jonmeow, zygoloid, josh11b
- Comparisons for integer and floating point types
    - if both operands are integer types, produce mathematically correct results
        - can be formulated as either no conversions apply, or promote both sides to a sufficiently large type big enough to hold the values on both sides
    - float vs. float
        - if you have Float(m) and Float(n) then m > n means every f in Float(n) is exactly representable as a Float(m), so comparisons and other operations promote to the larger type
    - float vs. int mixed comparisons
        - can start with the policy that we force the user to convert to a non-mixed type
        - will end up writing `0.0` when `0` would be more convenient
        - other option: if every integer value is representable in the floating point type, then promote the integer value and compare
            - could treat integer literals as promotable to float as long as their value is exactly representable even if other i32s can't
    - Summary rule: either you get the mathematically correct answer or a compile error
    - Could generalize these to other mixed operations, like `+`, etc.
    - Rules:
        - implicit conversions are never lossy
        - comparisons, if valid, always give mathematically correct results
            - we provide comparisons between `Int(n)` and `Unsigned(n)` for the same `n`
        - we never invent an intermediate type that is larger than the operands
            - as a consequence: it will never convert both operands, which may be a rule we want to generalize even for non-built-in types
    - Even if we have an implicit conversion from string to string_view, still don't want to use string_view comparison for string vs. literal comparison

        - allows us to have a simple rule for looking up operations based on just the types on both sides of the operator
- Need to make a new version of the open discussion doc to fix weird indentation issues

2021-08-12

- Attendees: jonmeow, zygoloid, josh11b, chandlerc
- enums
- bit packing
    - perhaps there is a bit-packed thing declared like a class, with members having types like `u1`, `i3`, etc., with the restriction that you can't take the address of any member
    - bit-packed types support & and |
    - want a way to conveniently name the bit-packed value with all zeros except a single 1
- newtype for validation and for units
    - can postpone, not in C++
- constructing a derived object #741
    - Concern: virtual bases
        - by accepting the restriction that you can't inherit from multiple virtual base classes, can make it so the constructor that initializes the virtual base isn't the derived-most type, but the derived-most C++ type
    - Result: from Carbon's perspective, subobjects are initialized in memory order, with the exception of the vtable ptr
        - Can reuse tail padding of an earlier field to store a later field, without danger of clobbering
        - gives freedom to the implementation, but only for initialization, not later assignment
    - Consider variation on option 2b, let's call it 2c
        - distinguish "Base or derived" vs. "exactly Base" (like Base.novirt)
        - question: if you have a Base.novirt, can you call methods that are declared virtual, but not pure virtual?
            - "exactly Base" interpretation: you can make the call, but it does not use the vtable dispatch
            - novirt interpretation: you can't make the call
        - question: if I have an "exactly Derived" pointer, can convert it to a "Base or derived" pointer
            - novirt interpretation: can convert a Derived.novirt pointer to a Base.novirt pointer, but not a Base pointer
            - consequence: a constructor returning "exactly Derived" must fill in the vpointer
            - finality applies to objects rather than only types
- josh11b opinions on finality: all classes are either base classes or final classes
    - abstract base class: base can't be instantiated, has pure virtual methods -- but Base.novirt can be instantiated

        - base class with virtual destructor: can be instantiated, pointers to base mean "base or derived"
        - base class non-virtual destructor: can't be instantiated, pointers to base mean "base or derived"
        - final class: can be instantiated, pointers to final mean "exactly final"
    - Conclusion: Like option 2b, particularly how it handles abstract base classes
    - Need to make `super` a keyword so we can reserve that field in the struct used to initialize the derived type
        - Need to allow `.super`, possibly other `.keyword` as well so we can access those fields from C++ types
    - Need to harmonize casing: perhaps all language-provided names should be lowercase?
        - `bool`
        - Example: `Self` should changed to `self`
        - Make a leads question
    - In 2b option, Base.novirt could store nullptr in the vtable pointer slot in both hardened and optimized mode
        - vtable pointer only needs to be set in base-most and derived-most function
    - Should look like other cases where we are using facet types, like `const`
    - Don't like the name `novirt`, maybe `base`? maybe `impl`? Unfortunately can't use the term `proto` (confusion with "prototype" and "protobuf")? Used as a prefix as in `base MyBaseType` or `impl MyBaseType`

```
base class MyBaseType {
// or if we don't want to put something before the introducer
// (which we might also do with `private`):
// class MyBaseType base {
  fn Create(...) -> base MyBaseType;
}

class MyFinalType extends MyBaseType {
  fn Create(...) -> MyFinalType;
}
```

    - Can only call methods that define their me-type as `novirt`, which allow you to share functions used during construction, statically validating that it won't call virtual functions
    - "base facet used for initialization"
    - Requires: all your base are belong to us meme in documentation
    - associated type syntax question #739
        - reason to specify the type (option 1), is when you want to use the same value for associated types from 2 interface implementations with the same name; want to specify the type as the intersection that satisfies the constraints from all the interfaces

        - more future proof
        - evolution is still fundamentally hard: any change either breaks impls or users
        - maybe make the type used by the interface available, via `auto` or `typeof(Interface.AssociatedType)`
        - Rule: type in type's impl is more restrictive than the type in the interface's associated type
            - To make the constraints on an associated type more strict, first change impls
            - To make the constraints on an associated type less strict, first change the interface
- alias proposal
    - associates two dotted names
    - Chandler doesn't like `alias A = B`, want something different from initialization or assignment
        - `alias A to B`; concern that order is ambiguous
        - `alias A <- B`
        - `alias A := B`
        - `alias A for B`
        - `alias A ~ B`
        - `alias A is B`; conflict with `3 is i32`
        - Everybody but Chandler likes `=`
        - will need to be a leads question

2021-08-10

- Attendees: josh11b, gribozavr
- Swift coherence
- Swift access control
    - no `protected`, but has file-private and module-private
    - derived classes can't implement private methods of parent, unlike C++

2021-08-09

- Attendees: chandlerc, zygoloid, josh11b, jonmeow
- `=` in struct patterns?
    - needed for defaults in function declarations, but don't need defaults in other contexts

```
var {x: Int = 0, y: Int = 0} = {.y = 2};
```

    - refutable patterns matching structs with a particular value for `x`

```
match (f()) {
  case {.x = 3, y: Int} => Print("(3, ", y, ")");
}
```

        - Can we combine these two? This would be legal if the thing after the `.x =` was a pattern:

```
match (f()) {
  case {.x = 3, .y = y2: Int} => Print("(3, ", y2, ")");
  case {.x = (3, n: Int), .y = 5} => ...
  ...
}
```

        - josh11b, mconst prefer only one level of destructuring; keeps to the readable subset, can do multiple destructurings if you need to do multiple levelspo
        - zygoloid is happy to try this, but is cautious because it's a departure from what most pattern matching languages do
    - It should be correct to switch between a function that consumes and an another overload that takes an immutable view
        - Needed both for refactoring and for optimizations
        - Could probabilistically switch between the two when running tests to validate that it is safe
        - Not expected to be a problem for users, based on our experience with similar things in C++
    - How should interfaces handle the distinction between parameter passing modes (eg, pass by immutable value vs consume vs ...)?
        - zygoloid: idea: we define a partial order over passing modes, ordering "a <= b" if we can correctly degrade from b to a. An interface specifies the minimum ordering that any implementation must be able to support, but impls can also implement stronger passing modes as overloads. A use of a generic can provide an argument supporting a stronger passing mode, which will (may?) get selected by monomorphization. Eg:

```
interface Comparable {
  // Use default passing mode (pass by immutable value).
  fn Compare(a: Self, b: Self) -> Order;
}
// A map implemented as a lazily-sorted vector.
class LazilySortedVectorMap(Key:! Type, Value:! Type) {
  var vec: Vector((Key, Value));
  impl as Comparable {
    fn Compare(a: Self, b: Self) -> Order {
      // Compute relative order without modifying a and b
    }
    fn Compare(consume a: Self, consume b: Self) -> Order {
      a.sort();
      b.sort();
      return Comparable.Compare(a.vec, b.vec);
    }
    // Maybe overloads for the case where one is consumable but the other is
not
```

```
  }
}
fn G[T:! Comparable](consume v1: T, consume v2: T) {
  Print(T.Compare(v1, v2));
  Print(T.Compare(~v2, ~v1));
}
fn F(consume v1: LazilySortedVectorMap(Int, Int),
     consume v2: LazilySortedVectorMap(Int, Int)) {
  G(~v1, ~v2);
}
```

Here, we can type-check `G` using only the `Comparable` interface, but when we come to monomorphize it for `T = LazilySortedVectorMap(Int, Int) as Comparable`, we can select the `consume` overload of `Comparable.Compare` instead of the pass-by-immutable-value overload, confident that doing so is correct.

- Consider this example:

```
interface ConvertibleToString { fn ToString[me:Self]() -> String; }
external impl i32 as ConvertibleToString { ... }
// Can say:
var n: i32 = 7;
(n as (i32 as ConvertibleToString)).ToString();
n.(ConvertibleToString.ToString)();

class Song {
  impl as ConvertibleToString { ... ToString ... }
}
// Expect this to be an error:
n.(Song.ToString)();
// Song.ToString is a method with me: Song
// ConvertibleToString.ToString has a more general me type
```

  - `Song.ToString` is not the same as `ConvertibleToString.ToString`, it has at least a different `me` type
  - Is `Song.ToString` equal to `(Song as ConvertibleToString).ToString`? Answer: Yes
  - Can't observe difference whether `me` is `Song` vs. `Song as ConvertibleToString`
  - Is it correct to say that every member of `Song as ConvertibleToString` is also a member of `Song`? Yes, if impl is not external. This includes members of `ConvertibleToString` that are not explicitly named in the `impl` definition but have defaults.
- Would like a type-of-type for a struct type that matches any struct type with the same fields in any order

    - Analogous to the type-of-type that we need to say "all types with the same representation as `Song`" for the "combining multiple sort orders for `Song`" use case
    - In both cases, want `=` and `==` to work broadly, and `<`, to work narrowly
- Prefer `private` or `internal`?
    - ChandlerC and Josh11b: `internal`, make `private` if needs linkage
    - zygoloid: `private` or `private internal`
    - maybe we shouldn't have two orthogonal concepts and tell people to correct from one to the other
    - Josh11b: my preference would be `private` would restrict access and automatically give internal linkage if possible, and there would be `private.internal` and `private.external` to explicitly specify the linkage

2021-08-02 part 2

- Attendees: chandlerc, josh11b, zygoloid
- Long discussion about name lookup, on-demand type checking, aliases, looking up vs. looking in the whole file, #472: Open question: Calling functions defined later in the same file
- Lambda syntax
    - Either `$` or `\` would reasonably introduce a lambda expression
    - `$($1 + $2)` would have two template parameters that would only get their type when used
    - Possibly allow lambdas with statements inside `${...}` as in `${ if ($1 < $2) { return $1; } else { return $2; } }`
    - Concern: need a way to specify parameter types for some situations
        - Maybe: `$(x: i32, y: i32) (x + y)`? Could detect this case by the sequence "`$ (` <id> `:`"
        - Maybe the body expression would be prefixed by `=>`? So `$=>($1 + $2)` and `$(x: i32, y: i32)=>(x + y)`
        - Leads to a uniform syntax, where there are a sequence of optional clauses, similar to `fn` syntax with a different start:
          `$[`<deduced params>`](`<explict params>`)->`<return type> `{ ... }`
          or:
          `$[`<deduced params>`](`<explict params>`)->`<return type>`=>(...)`
    - Bunch of questions about capturing; a bit different from deduced params
    - Something about passing captures along as long as the lambda doesn't escape the current scope? Something about using the type and the casts of that type guiding how the arguments are supplied to the lambda when called

2021-08-02

- Attendees: chandlerc, jonmeow, josh11b
- Some discussion of "breaking ties" being preferable to making overlapping blanket impls illegal

- Idea: using `break` to return a value from a loop-as-expression

```
var i: i32 = for (x in container1) {
  var j: i32 = for (y in container2) {
    if (y mod 17 == 3) { break y; }
  } else {
    break 0;
  }
  if (j != 0) { break j; }
} else {
  break 0;
};
```

  - requires the `for` to have an `else` (`on finish`? `on complete`?) with a `break` so that the loop always produces a value, or be a `while (true)`
  - can't generalize `break` for `if`, since we need `if` to decide whether to `break`
  - need some other ternary conditional expression
    - maybe: `if <condition> then <true result> else <false result>`
    - would we allow: `(if b then x else y) = 3;`? It would parse, and then it comes down to the type system
    - concern: do we expect a statement or expression when we see `if` at the beginning of a statement?
      - probably a statement, unless we have some other way to disambiguate
      - similarly `for` at the beginning of a statement introduces a `for` statement, similarly for `match` and anything else we want to allow to also be an expression
  - control flow useful in initializers, as seen in Rust; similarly for borrows
    - allows stricter preservation of type information

2021-07-30

- Attendees: josh11b, mconst
- mconst is sad about having to write the `U` in the parameter list in some cases, like so:
  ```
  fn Cast[U:! Type, T:! ConvertibleTo(U)](x: T, U) -> U
  Cast(3, i64)
  ```
  - The only reason this unintuitive pattern-match against U is required is because our syntax requires implicit parameters to appear before explicit ones. The need would go away if we just merged all parameters into one list:
    ```
    fn Cast(U:! Type, implicit T:! ConvertibleTo(U), x: T) -> U
    Cast(i64, 3)
    ```
  - Similarly, this would work fine if Carbon just did it the way other languages do, with a separate syntactic list of generic parameters:

```
fn Cast[explicit U: Type, T: ConvertibleTo(U)](x: T) -> U;
Cast[i64](3);
```

- Eventually we want to express higher-order (or is it higher-ranked?) types
  Given `interface I(T:! Type) { ... }`
  Want to be able to take a parameter with type: "Function taking a type `T` and returning a type satisfying interface `I(T)`"
  - Use case:

```
fn Search(takes this weird thing T -> Container(T), some other stuff) ->
Result;
class Queue(T:! Type) {
  impl Container(T) { ... }
}
fn BreadthFirstSearch(some other stuff) -> Result {
 return Search(Queue, some other stuff);
}
class Stack(T:! Type) {
  impl Container(T) { ... }
}
fn DepthFirstSearch(some other stuff) -> Result {
  return Search(Stack, some other stuff);
}
```

  - Do you end up writing `fnty (T:! Type) -> Container(T)`?
    or `fnty[explicit T: Type]() -> Container(T)`?
- Alternative generic syntax: `fn Cast[U: Type]<T: ConvertibleTo(U)>(x: T) -> U;`
  - [] = explicit generic, <> = implicit generic, () = explicit dynamic
  - What orders do we allow? () always last, of course
  - Write `Vector[i32]` following declaration `class Vector[T: Type]`
  - C++ approach of compiler figuring out explicit vs. deduced should be on the table since it is at least very familiar, could then just use `[...]` for generics
- Concern about deducing T from a list of type T when the list is empty, may lead to wanting to explicitly specify a type parameter that is usually deduced
  - Example: Empty variadic list for Sum where we want to know the type of the zero return value
  - Example: Variadic list where we don't care about the type when the list is empty, but the compiler is likely to complain that the type might possibly be used in the function body

2021-07-29

- Attendees: jonmeow, josh11b, zygoloid, jorgbrown, chandlerc
- Do we need an introducer for anonymous data class type expressions?
  - Conclusion: No

    - We'll call them "structural data classes" which we can abbreviate "struct" informally
    - We don't need an introducer if we aren't introducing a name
    - `{}` will represent both an empty struct value and empty struct type
    - Performing the same experiment that we are with tuple types
    - Better to try things that are risky but are more desirable if they work out first
- Arrays
    - Considered array literals of the form `[...]`
    - No corresponding array type expression that we like without an introducer
        - considered a number of options, e.g.:
            - `MyType[4][3]` indexes don't match order when have multiple dims
            - `[3][4]MyType` dims are always left-to-right, before name for type, after name for access, starting with `[` ambiguous if we have array literals, but maybe if we don't?
            - Rust `[..; ..]` syntax not appealing, same ambiguity starting with a `[`
        - expect it to be a library type, don't expect it to be as common as tuples and structs and will have more variations (like `std::vector` with dynamic size,)
    - Concern: difference from tuples is that they would all be the same type, but really what we want is *convertible to the same type*. So we would be fine initializing an array of `Foo` objects with a tuple with varying types, as long as they are all convertible to `Foo`. Better than trying to convert them all to the same type preemptively when constructing the `[...]` expression, before we know that the common type should be `Foo`
    - Conclusion: no array literal syntax, instead use tuples
        - takes the pressure off for coming up with an array type expression syntax without an introducer
- Also had a long conversation about what type the subrange of an array is
    - Don't want separate types for "reference to array of length 3" and "reference to a part of an array of length 3"
    - Conclusion: Rust's approach is a little weird, but seems good since it solves this problem
    - Main consequence is that there are two array types: "arrays" with statically known size and "slices" with unknown/dynamic size
    - Means supporting dynamically sized types (DSTs) more generally
        - Probably wanted to for other use cases anyway (for example, non-final class types)
        - Common enough pattern to have an array at the end of a C struct with dynamic size
        - Rule: local variables and arrays can't have DST
            - Can have a local variable of fixed-size *storage* type, and can allocate a DST inside
        - Rule: types can have at most one DST field as long as it is last

            - Having a DST field makes you a DST
            - Requiring there to be at most one and last means avoiding dynamic field-offset computations, which would require the size available dynamically
        - A DST won't implement the `Sized` interface. `Sized` will be required by common interfaces like `Copyable` and `Movable`. However if you only deal with pointers to a type, can use `Type` without a `Sized` requirement
- Tuples should implement all the same interfaces as structural data classes, so since tuples aren't classes we should probably call the associated interface `Data` not `DataClass`
- Build
    - Can we use map-reduce like approaches to make the monomorphization efficient enough to use generally for fast development build modes, to simplify compilation?
    - Lots of different strategies for making fast "time to build and run one test" for development, including interpreters, jit, and non-monomorphization of generics. Not sure if any are worth the development cost / compiler complexity
- Specialization
    - Could we just use a numeric priority to resolve ambiguities from blanket impls to avoid compiler errors from overlap rules? Would be nice to avoid library conflicts, particularly when either blanket impl would work.
    - Only a problem for blanket impls

2021-07-27 part 2 on access control

- Attendees: josh11b, chandlerc, zygoloid
- (Some discussion about Carbon: access control using facet types (v2) first)Should support `internal` in addition to `private`.
- `public`/`private` is about access, "can I call/name a thing"
    - `private` restricts to the same innermost enclosing lexical context (e.g. class or file; or maybe library or package instead of file?)
- `external`/`internal` is about reachability
    - `internal` restricts to the same library
- Consider `inline`, we have an inline function
    - In its body, wants to call some implementation-detail helper function
    - Those helper functions are reachable, so they have linkage
    - So the helper functions are `private` but not `internal`, since they are reachable but not accessible
- Reachability is about separate compilation, of which linkage is a part
- In this example `internal var x: Int`, the name `x` is not salient from a linkage perspective
- Proposed rule:
    - Any inline definition is only allowed to reach non-internal names

        - Any non-inline definition is allowed to reach internal (and non-internal) names
        - For purposes of this rule, templates would be considered inline, but it is actually more complicated, C++ got this right
            - Don't have to deal with ADL, just delayed member, overload, and impl lookup
    - If we could eliminate `inline`, then we wouldn't need both `private` and `internal`
    - Rule is: you want to make something restricted, mark it `internal` unless the compiler tells you it is reachable and so you have to make it `private`.
    - Consequence: can have private things that are not members
        - need for durable module boundaries in the presence of `inline`
    - Observations:
        - It is exceedingly rare to need both `private` and `internal` on a single declaration
        - For non-member, never need both
        - If you have inheritance, and descendants defined in the same library and they should not have access, then you might use both `private` and `internal`
    - zygoloid: often finds it useful to have encapsulation boundaries that are smaller than a file, sometimes even smaller than a class
    - can still audit / have control over code in the same file
    - chandlerc: would only use `private` together with `internal` in the specific case where you want to restrict access to a unit smaller than a library.
    - C++ ensured that this code would compile, even though `Ft` is not in the same library as `internal x`

```
fn Ft(template T:! Type, thing: T) -> Int {
  return thing.x;
}

interface Gish {
  fn G[me: Self]() -> Int;
}

fn Fg[T:! Gish](thing: T) -> Int {
  return thing.G();
}

// we import `Ft` and `Fg`, they come from far away

class S {
  private internal var x: Int;

  internal impl Gish {
    // (I hope `internal` below is redundant w/ above?)
    internal fn G[me: Self]() -> Int {
```

```
      return Ft(private internal Self, me);
    }
  }
}

internal fn UseGish(s: S) -> Int {
  return Fg(s);
}

internal class InternalQ {
  var x: Int;
}

internal fn UseInternalQ(q: InternalQ) -> Int {
  Ft(InternalQ, q);
  var heap_q: UniquePtr(InternalQ) = ...;
}
```

- chandlerc: `G` , `Ft`, `UseGish` could compile by the following logic:
  - `G` not inline so it should have access to internal `x`
  - will instantiate the template when compiling `G`
  - could let `Self` have internal access in the body of `G`, could be represented by an "internal facet" which is an internal type
  - Since `Ft` is instantiated with an internal type, its instantiation is also internal
  - C++ committee studied this problem and concluded that they expected templates to work this way
  - josh11b: concern this is much harder to explain than the simple story "internal things are only accessible within this library"
  - chandlerc: should be able to call templated function parameterized internal types like `InternalQ`
- zygoloid: concern with `private` at namespace scope might be in conflict with facets for access control
- chandlerc: facet types allows us to model the parameterization of templates and generics on the type
  - In C++, one of the big challenges here is how do you look at your template and determine if it is internal; can use whether any parameter is internal to determine if the template instantiation should be internal as well **Chandler, please update**
- Concern, what if we change the code, moving the declaration of `x` after the definition of `G`?

```
fn Ft(template T:! Type, thing: T) -> Int {
  return thing.x;
```

```
}

interface Gish {
  fn G[me: Self]() -> Int;
}

fn Fg[T:! Gish](thing: T) -> Int {
  return thing.G();
}

// we import `Ft` and `Fg`, they come from far away

class S {
  internal impl Gish {
    // (I hope `internal` below is redundant w/ above?)
    internal fn G[me: Self]() -> Int {
      return Ft(private internal Self, me);
    }
  }

  private internal var x: Int;
}

internal fn UseGish(s: S) -> Int {
  return Fg(s);
}
```

- Now the instantiation of `Ft` needs a *complete* facet type of `private internal Self`.
- Assuming template instantiation is immediate vs. delayed
- Josh11b: you used `Self` when it was incomplete, perhaps we need to require G to be defined out of line, when `Self` is complete
- ChandlerC: Look only up rule for name lookup interacts with these choices. Members are in Carbon, unlike C++, written qualified inside member function bodies.
    - Makes it easier to always go up, even in inline method bodies, will find `me` and won't go anywhere else
    - Can't typecheck method definitions written lexically in the type's declaration scope since `Self` is *always* incomplete there.
    - If we delay type checking, makes it harder to implement the rule that name lookup always looks up
    - Would do unqualified name lookup at the time it is parsed, not during type checking (where you (must) do member lookup)
- ChandlerC: made uncomfortable with lexically inline method definitions
    - rules seem awkward
    - maybe we require all methods to be defined lexically out of line?

- Deferred type checking might be okay because of accessing members through `me` means that the members of `Self` are only looked up at the time of type checking when the full definition of `Self` is available.
- Is this a warning against facets?
- What if we allowed you to declare internal methods lexically out of line?
  - could be missing from API file
  - would largely mitigate the cost of insisting on methods defined lexically out-of-line...
  - public facet would still be a complete type
  - doesn't raise problems of extension methods until we use facets to model the "internal" view -- this starts to break down
  -

```
class X {
  constexpr static int ComputeLength(int x, int y) {
    T sum = x + y;
    return sum;
  }

  constexpr static auto my_member = ComputeLength(1, 2);

  int my_array_member[my_member];

  using T = int;
};
```

```
class X {
  var i: Int = 42;

  // Is this allowed?
  var j: Int = i;

  // What address is this?
  var p: Int* = &i;
};

fn MakeMeAnX() -> X {
  returned var x: X = {};
  x.i = 1
  x.j = x.i;
  x.p = &x.i;
  return var;
}


let a: X = MakeMeAnX();
```

```
// a.p dangling pointer

class LL {
  var i: Int;
  var n: LL*;
  var p: LL*;
}

fn InitLL(Int a) -> LL{
  returned var b: LL = {.i = a, .n = &b, .p = &b};
  return var;
}
```

2021-07-27

- Attendees: josh11b, chandlerc
- Generic syntax, rejected alternatives
  - fn CastAVector[T: Type](v: Vector[T], generic DestT: Type) -> Vector[DestT];
    class Vector[T: Type] { ... }
    - whether a parameter was explicit vs. deduced inconsistent across function and class declarations
  - fn CastAVector<T: Type>[DestT: Type](v: Vector[T]) -> Vector[DestT];
    class Vector[T: Type] { ... }
    - [] - explicit and generic, <> - deduced and generic, () - explicit and dynamic
    - Concern: `Vector[T]` used in the same contexts as `[...]` used for indexing
    - Note: Go using [...] for generics (as of sometime Jan 2020..Jan 2021, but in the accepted proposal)
    - was not considered
  - fn CastAVector[T: Type]<DestT: Type>(v: Vector<T>) -> Vector<DestT>);
    class Vector<T: Type> { ... }
    - problem: angle brackets in type expressions
  - fn CastAVector<T: Type>(v: Vector(T), <DestT: Type>) -> Vector(DestT);
    class Vector(<T: Type>) { ... }
    - was considered
    - not trivial to parse, but doable
    - too much punctuation
    - nice that <...> is associated with generics, but with enough differences to be concerning
  - fn CastAVector[T: Type](v: Vector(T), [DestT: Type]) -> Vector(DestT);
    class Vector([T: Type]) { ... }

        - no reason to prefer this over previous option, since <...> more associated with generics than [...]
    - Brainstormed a bunch of different spellings of the `:!` position
        - <id>: Type
        - id:# Type
        - id:<> Type
        - id: <Type>
        - generic id: Type
        - No specific objections were discussed, but none were sufficiently appealing compared to `:!` to really warrant driving more discussion. Generally happy with `!` being associated with "compile time"
    - Never really broke out of the idea that `[...]` were for deduced parameters, and so didn't really consider `Vector[Int]`.
    -
-

2021-07-26

- Attendees: josh11b, zygoloid, jonmeow, wolff
- Basic stuff we are missing for writing code
    - Integer types
        - To start: built in
        - bool is not an integer type -- see Chromium bug confusing `&` and `&&`
        - only widening implicit conversions
        - `and` and `or` instead of `&&` and `||`
            - `and` and `or` are easier to type than `&&` and `||`
            - generally nice to use keywords for control flow
        - question: use `not` instead of `!` for consistency?
            - but consistency with `!=`
            - keyword would be more closely associated with precedence?
        - don't want to use `^` for bit-xor, too rare for a dedicated single-symbol
        - other bit operations?
            - being taken seriously as a low-level programming language
        - Spelling
            - Don't like `I32`
            - Options: `Int32`, `int32`, `i32`
            - Swift: `Int32`, Rust: `i32`
            - Swift has `Int` which is either `Int32` or `Int64`
                - Rust has `usize`, but mistake to assume bits to store an allocation size is the same as bits for a pointer

- Chandler doesn't want to expose pointers as integers (at least in safe code)
    - concerns: provenance, secret security bits
    - still allow (maybe unsafe) pointer arithmetic, storing integers in the low bits of pointers
- ~~May~~ want a signed `ISize` or `Size` for maximum allocation size and max array size
    - Want to support both (embedded) 32 and 64 bit platforms
    - Question: do we need the size type to be unsigned 32 bit, to access 4GB instead of just 2GB?
    - Choice of using unsigned in C++ has lead to problems since (signed) `ptrdiff_t` is not big enough to store offsets into 3GB objects (used by databases)
    - Similarly, like being able to store -1 for `string::npos` without danger of it being treated as a large valid position
- Default `Int` type; if defined, expected to be used a lot
    - Rust option: no default size, but use `i32` to be concise; zygoloid likes this choice
    - Perf: Pick a default for performance: `Int` == `Int32`; matches C++, Java, C# in practice for migration; `Int32` still available so you can say "I actually meant 32bits"
    - Safety: Pick a default for avoiding overflow: `Int` == `Int64`
    - BigInt: `Int` is an arbitrary precision type, probably not appropriate for a performance language
    - Platform: `Int` could be `Int32` or `Int64`, based on the platform, like Swift; don't like this option, prefer to use `ISize` for when you specifically want a platform-specific size rather than by default
- Comfortable with "Perf" option, provisionally, spelled `Int`, `Int32`, etc.
- Unsigned types use cases:
    - I wish this was 1 bit wider and I don't care about representing negative numbers (database on a 32 bit platform using all 4GB of memory, packing as many fields as possible into as few bits as possible in a struct)
    - I want to make invalid states unrepresentable, and negative numbers are invalid -- if supported, should support it more broadly with more arbitrary ranges?

        - modelling modular arithmetic % 2^32, % 2^64, etc. (hash functions, PRNGs, compression algorithms, crypto)
        - not representing a number, instead manipulating bits with bitwise operators: bit fields, bit masks, etc.
        - hash function
        - I want to perform some clever hakmem algorithm using bit manipulation tricks on a number (count bits, log2, FFT, binary GCD, etc.)
        - C/C++ interop
        - Interop with file formats and network protocols (eg, representing IP4 addresses as 4 unsigned 8-bit integers)
        - I want to perform a calculation that may overflow and I don't want undefined behavior if it does, but I don't care about the answer
        - I want to perform a calculation and detect if it did overflow
    - Would like to support overflow use cases more directly with library support
    - Would be fine if clever bit manipulation tricks required bit-casts, just like you would expect with fast inverse square root
    - Do we want different types for different use cases?
        - Concern is different behavior on overflow `Mod64` vs. `UInt64`
        - Swift instead uses different operators for wrap-around
        - Previous discussions favored "overflow is an error" and other overflow behavior provided by libraries
        - Probably need to make this a question-for-leads issue
    - More musing about spelling int types
        - previously discussed in #543
- Non-English native coders?
    - will still follow most languages using English keywords
    - avoid English linguistic things like Python's `is not` and `not in` operators
    - comments must support all languages well
    - follow unicode recommendations for identifiers to allow native
        - concern is confusables, will use linter to allow it to evolve quickly
        - maybe per-file list of scripts used
- Assignment and initialization operations
    - chandlerc asking about initialization
    - chandlerc asking about assignment
    - open question added to struct proposal
    - disagreement about assign each field vs. destroy & init by default

    - if left and right have different field orders, use destination field order since it will work for classes as well

2021-07-19

- Attendees: josh11b, chandlerc, jonmeow, zygoloid, mconst
- What goes in the type?
    - The type has to have the complete description of a value's behavior/capabilities
        - chandlerc: restrictions can come separately, it's asymmetric
        - things not in the typesystem have a way of worming back in
            - example: noexcept
            - history: old-style exception specification not part of type
            - but function pointer/argument assignment would have a ("non-type") check that the exception specs are compatible
            - problematic case: conditional expression with different exception specifications on both sides
            - ended up needing all the typing rules
            - C++11 added noexcept, added new style
            - around C++14 or C++17 made it part of the function type
            - both Clang and GCC have a flag to opt into the old behavior
        - KDE is the example for what is needed to preserve C++ ABI compatibility
            - historically had dynamic link slowness problem
    - Inconsistency is awful
- `class` vs. `struct`
- data class
    - opt in for type
        - Does it list all the interfaces explicitly, or is it just "all that apply"?
        - leaning toward the latter
    - default implementation for interfaces provided with interface
        - going to have to be a templated implementation
        - needs a constraint: get the type tuple for the data members and then a constraint "all types in this tuple implement this interface"
        - general metaprogramming approach may give better errors
        - or just "all member must satisfy constraint X", which would also give good errors
    - a way to implement an interface in a "memberwise auto" way
        - define it for a pair assuming it is defined for each member
            - problem: can't distinguish (int, int, int), ((int, int), int), (int, (int, int)) and want them to hash and serialize differently
            - serialize needs metaprogramming, might have a simpler thing for easier cases
    - more generally want to be able to say "this type has tag X" and interfaces can provide implementations for types with tag X

        - one possibility: tag a type by saying it implements X, which is an empty nominal interface
- don't use overloading, use an `if` on the type
    - we only support case where all function signatures are the same, so type checking does not depend on which overload is selected
- don't want absolute addresses in vtables and witness tables
    - Bunch of interesting tradeoffs in the implementation space here due to PC-relative / IP-relative addressing and PIC code (for the sake of ASLR even if not for plugins / loadable shared libraries)
    - Also interesting to thing about CFI based lowering strategies here
    - Big thing is that we want to make sure we don't accidentally leak raw function pointer based implementation of dynamic witness (or Carbon-ABI virtual) tables so that we can swap out different, interesting implementation strategies
    - A huge element of this is that we want to hew towards the more DLL-style model of fully separate programs that just happen to be loaded into the same (virtual) address space and are able to share memory easily.
        - Likely want calling between plugins / dynamic libraries to really *be* an FFI more than something like a dynamic library call on C++/ELF
        - Gives a smooth ramp towards IPC or even RPC without sharp edges
        - Really move towards a language-builtin protocol system.
            - General purpose language facility
- Like public by default for class members, what about in API/impl files?
    - default to external for both
    - impl file must match api files for anything external
    - maybe `private` == `internal`?
        - concern about linkage around friends
        - concern about linkage and templates calling internal functions
    - C++ made `inline` (even more) magical - inline can't touch static, templates, static variables with const initializers, and so on
    - maybe we could make things as if visible from a linkage perspective
        - concern around compiler not being able to prove things internal, e.g. if there is metaprogramming
    - guidance: use `internal` until you get a compiler error that tells you to say `private`
        - MConst: maybe access control would be advisory, works well in Python; if people need to do it is it better to make people do crazy casts
            - Big companies are different than game companies
            - Librarians say they need access control with teeth for evolution; enough that people usually don't ask for rollbacks when they bypass it and private details change

2021-07-12

- Attendees: chandlerc, jonmeow, josh11b, zygoloid, mconst
- Will make Carbon: Low context-sensitivity principle into a proposal
- Struct proposal
  - Multiple paths forward for mixins, big questions are whether a mixin is its own type that is a member of the containing type, and whether they are templates
  - Clang has one example of multiple inheritance without virtual, but still not motivational to support multiple inheritance in Carbon
  - Make `DerivedPtr` derived from the interface-as-base class for C++ interop instead of making `DynPtr` derived from the interface-as-base class automatically
  - Three choices for destructors with inheritance without virtual destructor
    - Avoid non-trivial derived destructors, use unsized delete
    - Can only dynamically allocate and delete final types
    - Distinguish between a pointer to "exactly base" (default) vs. "base-or-derived"
  - Can make the last two choices equivalent by for every non-vtable non-final type creating a corresponding final derived type that inherits from it and adds nothing
    - instead of having two kinds of pointers, have pointers to two different types
    - implicit conversion from final to non-final
    - may allow an explicit downcast from non-final to final
  - Question: local variables
    - are they the final type or the non-final type?
    - are they implicitly the final type, so you don't have to spell the longer final type name?
    - vector<T> also wants to hold the final type
    - argument for distinguishing the pointer instead of the type; only time we care about this difference is when we have a pointer
  - Pointers that can delete the object (unique_ptr/box) vs. pointers that need a lifetime annotation
- We agree to say something short like `box` instead of something long like `unique_ptr`
- Some discussion re: in-person vs. VC
- Struct use case naming:
  - data classes
  - encapsulation
    - no inheritance - the default case
    - inheritance with subtyping
      - non-polymorphic (high-performance data structure libraries; example: sentinel node in linked list, base has next/prev, derived has data, lots of code can operate on the

base; typesystem verifies that methods like splice, etc. don't touch the data; also map & tree but it is more complicated; have some extrinsic way of knowing when it is safe to downcast or don't downcast) - in generic code where a prefix is known to avoid monorphizing/instantiating, but extended generically
        - polymorphic (opt-in on a method level)
            - interfaces as base classes
    - mixins
    - Interop with C++ multiple inheritance - maybe we don't support interop with iostreams, we just use wrappers
- public vs. private
    - When encapsulating, would be nice to have default public methods to avoid clutter for readers, and default private data since that is the common case
    - Would be nice to have public then private divide, but that is awkward if we want to allow dependency orders
    - MConst likes to be able to look at an individual member and see if it is public or private, particularly for very large class declarations
        - ChandlerC worries about C++ programmers looking at declarations and complaining about Java ceremony
        - MConst: people don't complain about the `pub` markers in Rust
        - Do need to be mindful of the total declaration length
        - Python, Kotlin: public is the default for everything
            - makes the public API is easy to read
            - small & simple things are small & simple; if your class has even one private thing it isn't simple
            - function declarations are more complicated, less bad to have "private" on variable declarations which are much simpler
            - smooth path from data class to encapsulation
            - MConst likes this approach in general; simple, matches real world use cases well; even Kotlin from SmallTalk tradition went this way
    - Going to make public the default
        - no `class` introducer, just `struct`
        - maybe `base` or `derivable` to opt-in to non-final like `struct Name base`
        - ChandlerC: doesn't like annotation syntax here since it affects the legal syntax inside the struct declaration; would have to make virtual also an annotation
    - mandatory `override` in subclasses, must go in same place as `virtual`
    - inheriting overloads
        - avoid unexpected behavior of adding something with the same name as a base hides the methods in the base

        - don't have the multiple-inheritance problem where two parents have virtual methods with the same name
    - need to handle contravariance in parameter type, in that case it should shadow base function (comes up for both virtual and non-virtual)
    - solution: ask the programmer to make a choice
    - don't want to have virtual dispatch happen prior to overload resolution (even more important for generics and templates where it isn't necessarily a finite set)
    - if you define anything with the same name as a virtual method in the base, the base virtual method must be overridden in the derived type; can have syntactic sugar like C++'s `using` to import base names into the derived type
        - important so that overload resolution is always coherent
    - use case like assignment operators where the base operator = takes a base and want to hide the base in the derived to use derived operator = taking derived instead
        - maybe assignment is special, want the same behavior as destructor
            - maybe constrain the implementation of the assignment interface to only pointers that have ownership? this is a strong requirement
        - if you have non-polymorphic assignment, must have the types match
        - could also come up with a user-defined function that follows the same pattern
        - maybe we don't support shadowing for non-polymorphic methods? just special-case assignment
        - hash table with an update method, derived hash table that also wants to update? maybe this is a bad case of derivation
            - if you are not extending, then you don't have a subtyping relationship unless you make the methods polymorphic
            - pushes us further in the direction of non-polymorphic inheritance is a weird, advanced thing
            - need to emphasize how breaking subtyping is dangerous in C++, and describe the alternative
    - going to enforce stricter Carbon rules on the Carbon side of C++ interop
    - assignment is still scary, danger of slicing, can we not support hiding with assignment?

2021-07-08

- Attendees: chandlerc, josh11b, jonmeow, gribozavr, zygoloid
- Type-based access control
  - Carbon: access control using facet types
  - Goal: Handle delegation to generic code that needs specific access.
  - Non-goal: No need to support friendship across packages. If something cross-package is needed, handle it with visibility in the build system instead. This restriction is important for layering and the ability to name your friends.
  - Does not need to be bullet-proof. Okay if there is an escape hatch.
  - Possible goal: Would be nice to keep private details out of the exported public API of a library, so that implementation details can change without needing to recompile reverse dependencies.
  - Would be good to preclude overload sets with different access
  - Fine to reference a forward declared name in the same package.
  - Always name an entire scope as a friend, not something more conditional
  - Type parameterization may be used to pass the facet through
  - Name of the private facet for type `X` is `private X`, but can only be used if declared `friend` in the declaration of `X`. `X` names the public facet
  - `friend <name>`, `<name>` must already be declared
  - Fixes invisible friend mess
  - Should the private type be named `Self` or is that too error prone? Concern that another name will require a lot of boilerplate
  - chandlerc: Self in the signature should not give any private access, but the type of me in the function body should be the private facet
    - use `X` and `private X` to explicitly specify one or the other
    - main concern is `Self` in an output position; if that is ever a private type, could get access to that type via the type of the (pointer to the) method, using type deduction or reflection or something
    - lock & key pattern would have to explicitly write `private X` as the return type
    - Concern that `Self` in return type and `Self` in the function body mean different things, though this matches the behavior that you can access private member in the body
    - `me: X` and `me: Self` would behave differently, `Self` is what is special, `X` would not give you private access to members of `me` without a cast
    - Factory function returning `Self` can initialize private members without a cast, needs a cast if factory function returning `X`
    - Values of type `Self` change their type upon entering or exiting a function body?
    - Concern it would still work with complicated nested type situations?

        - For public functions using `Self` in the signature: declaration is parsed twice: As if a function forwards a function taking public Self to another function that accepts private Self values
    - Use case: writing a member function without private access; could use `X` instead of `Self`
    - Everything in the class declaration is included in the private facet; things that are public get a variant in the public facet with Self changed
    - Linked list node with a public next pointer: privateness of Self will be inherited
    - Just public forwarding are what are exposed at module boundaries. No private names, just size and alignment are exposed
- What is the default access control?
    - C++ reality is a gradual spectrum, rather than a sharp divide between encapsulated types and Kotlin-style data classes
    - maybe `struct` is all public - no access control, `class` is default private
    - transition from `struct` -> `class` when you want to add private members painful if this affects forward declarations
    - single main introducer that declares a user type independent of other distinctions like access control
        - don't want different kinds of types to be mangled differently
        - maybe additional modifiers at the end of the declaration or in the body
    - concern about ergonomics of both C++ and Java's approach, Kotlin does some interesting stuff in this space
- Different kinds of constants
    - 1. Initialized to a dynamic value but won't change afterwards
    - 2. generic constant, value not available at typechecking time, but known at codegen time
    - 3. template constant, value available at typechecking time
    - 4. alias: just about names; for example could have an alias to a namespace which is currently being added to.
    - There may be a distinction between dependent and non-dependent (like 10) template constants, both available after instantiation, but one is also available before, when typechecking a template before instantiation
    - Question: how many different kinds of syntax for declaring constants shall we have?
    - Josh would like there to be fewer syntaxes, and have the result depend on the nature of the right hand side of the `=`
    - Expressions are not decorated differently. If we write `A + B`, have to try and evaluate it to figure out what kind it is
    - Probably two different things to do:
        - Need a particular thing in a particular context, see if an expression is a valid example of that kind of thing. Example: here we need a generic constant, evaluate this expression symbolically.

        - given an expression, figure out which category it happens to be in, and then treat it differently. basically infer the "kind" of constant based on evaluation / other properties
        - the first of these is easier to describe rules for because you know a-priori what you're trying to accomplish. you try to do it, and reject if it fails.
        - second may be trickier because of fallback paths
    - [josh11b] but maybe for the second we just have something closer to a typesystem that reliably answers these questions. if you start with a template and add it to a generic thing, then it's a generic thing. etc.
    - [zygoloid] what if we have a condition? we need to evaluate the condition to know what kind of thing we have.
    - [josh11b] typesystem rules seem to work -- if we can evaluate the condition we know what to do?
    - [zygoloid] you're still doing evaluation, not just type checking.
    - [chandlerc] bigger difference than we're really getting at.
        - if (template) then template else dynamic
        - may need to recurse if the `if` condition also has a condition in it
        - makes it more like evaluation than type checking
        - can't bound when it will complete, like evaluation not type checking
        - unsurprising for users in the easy case, but very surprising outside the easy cases; misleading since easy cases work really well
        - saying what kind you are looking for ahead of time makes the hard cases easier
    - Example of surprise: "2^n - 1" works until `n` == 32, and then it compiles without error but you end up with a different kind of constant since it overflowed. No one reviewing the code expected a test, since it was a compile-time constant they expected a compile-time error if there was a problem.
        - https://godbolt.org/z/o18jY9Trj
        - `int64 cache_size = 1 << 32;` <- a more boring form that chandlerc wrote
    - [josh11b] But, maybe we don't want to have 5 different declaration contexts?
    - [chandlerc] Can figure out dependent vs. non-dependent for template constants just from the structure of the initializer expression
    - [zygoloid] well.... i think you still get into a mess actually
        - you can have this propagated in some cases
    - [chandlerc] but we can be conservative and eager, and that makes this much less problematic
    - [zygoloid] yes, there shouldn't be cases where we can't just propagate dependence when needed.
        - only benefit of non-dependence is better/earlier compiler diagnostics
        - is that part of the language rules? or something more vague?

        - if the template has no valid instantiations, should we be required to reject
    - [zygoloid] in any case, this won't change the number of different kinds of declarations
        - impacts the space of invalid but undiagnosed programs
        -
        - [josh11b] my rule is that, if you wrote a template that's your problem
        - [zygoloid] that's C++'s approach, but does come with portability challenges
        - [josh11b] solve this problem with generics?
        - [zygoloid] seems like generics solves a somewhat different problem space
        - [chandlerc] generics just solve this by happening to not have dependent code
        - [josh11b] and eliminating dependence seems good because its not something users really see or focus on. seems largely a problem for compilers and not pressing in a world w/ generics
        - [zygoloid] no, decelaration matching
            - you have a method declared inside a class template you want to redeclare it later on. which means we need to be able to look at two different declarations and say are these declaring the same thing? they're allowed to be syntactically different in certain ways and that means you need to do type checking and some semantic analysis on the template without instantiating it. you need to understand the methods it declares.
            - We can take a more syntactic approach that simplifies this, but it doesn't seem like it will completely solve this
    - [chandlerc] maybe bring this back to constants and their declaration contexts
    - [josh11b] yeah, templates are still pretty unspecified
    - [zygoloid] also declaration matching
    - [chandlerc] hypothesized: have three kinds of constants, which are reasonable for programmers to distinguish though the ergonomics may not be ideal, plus aliases which are actually fundamentally different
        - dynamic, generic, template constants; constraints on what you can write in the initializer
        - most commonly desired case will be integer constants which should be template
        - can we make a generic integer useful as an array bound?
    - Do templates have a cost even for constants in a function body?
    - [zygoloid] Only need to distinguish at interface boundaries. Inside a function body, everything could be a `let` binding
        - Need to know whether to specialize
        - Should we allow a generic constant in a loop body?

    - [chandlerc] `if constexpr` analog will I think cause problems
    - [zygoloid] `let n: Int = 5`
        - can use `n` as an array bound since when used we can look and see that it is the right kind of constant
        - or, can say that we can't use `n` in an array bound because `n` is not the right kind of variable to use in that kind of constant
        - either option seems to work
    - [chandlerc] Will the problems that occur outside of a function still affect things in a function body? 2^n - 1? How many steps to evaluate?
        - inconsistency!
    - [zygoloid] Consistency is interface boundaries vs. everything else
        - [chandler] interface boundary of `+`?
        - [zygoloid] interface of `+` does not require a constant, `+` implementation is not specialized on the arguments
    - [zygoloid] As long as we give an error instead of silently change behavior, not terrible
- [josh11b] are aliases yet another kind of constant?
        - aliases could be about names, or they could be like macros
        - they are binding a name to a piece of resolved AST, using the results of name lookup in the location where the alias appears
        - difference is whether only names are allowed on the right hand side
        - If we wanted to support these entities on the right hand side of an = in a `let` expression, we potentially wouldn't need a separate `alias` statement
            - names of overloaded functions
            - names of namespaces
        - another possible difference between `let` and `alias` is whether you state the type
    - [zygoloid] types are values, so nothing special there
    - [zygoloid] how would we model an overloaded function?
        - [josh11b] given that I want them to be closed, could represent them like a C++ singleton zero-sized object with an overload set of operator `()`s
        - [zygoloid] my idea is essentially the same: a function name is a value with a type that is unique to that overload set, and that type implements one interface for each overload.
    - [josh11b] Do we want to support overload sets as entities in their own right in Carbon more generally?
        - [zygoloid] yes, I think so, since there are other use cases like functions taking overload sets
        - [zygoloid] then we can use `let`
    - [josh11b] Problem with namespaces is they are open to extension and so don't have a fixed set of members
        - [zygoloid] do we want to introduce `alias` just for that one use case?

        - [josh11b] concern is not having an edge case that makes large scale namespace renaming difficult
    - The cases where we *need* alias support are names that are not values
        - the cases where it is more of a nice-to-have is when you just don't want to restate the type
        - or where you prefer to not think of the value having a type most of the time (eg, an overload set)
    - Is a namespace a thing, for example could it be passed as a template parameter?
        - openness of namespaces is a definite problem here
            - affects which namespace members would be visible in the instantiation
        - Can't say we have all the members in scope before we instantiate, unlike any other type
    - Is a namespace a weird thing that you can still use in a `let` but not as a template parameter?
        - maybe it is just a "symbol", the symbol document was an attempt to make names into things
    - `interface I { method F...; }`, can I write `alias IF = I.F` and then `x.(IF)();`?
        - [zygoloid] would like the answer to be: yes if and only if you can write `x.(I.F)()`
        - is `I.F` something that can be on the right hand side of a `let ... =`? It isn't even a function, it is an unbound method name
    - [zygoloid] `struct S { var n: Int; };`, can I write `alias Sn = S.n;` and then `s.(Sn)`?
    - [zygoloid] Since `alias` is about binding a name to the results of resolving another name, we can write `alias Sn = S.n`.
        - In contrast `S.n` does not represent a value that could be on the right hand side of a `let`.
        - Want `struct Pair { var first: T; var second: U; }; struct MapValue : Pair { alias Key = Pair.first; alias Value = Pair.second; };` to work.
    - Conclusion: we want both `alias` and `let`, and `alias` is specifically about names, and is needed for names that aren't themselves values, like namespaces and members of types and interfaces
- Back to constants

```
fn G(n: Int);
fn F(template T:! Type) {
  // Okay in template definition, and in instantiation
  let x: T = {.n = 0};
  G(x.n);
  // What about the remaining part?
  let U: Type = T;
  let y: U = {.n = 0};
  G(y.n);
```

```
    let C: Comparable = T as Comparable;
  }
  struct Z { var n: Int; };
  fn Run() { F(Z); }
```

- [zygoloid]
  One approach: `template` means that `T` has a more specific type than `Type`, that is unknown (and dependent) until instantiation
  - Consequence: `U` is different, since it actually has type `Type`
- Would this be different if `let U: auto = T`?
  - In the more specific type model, it would be different, and would compile
- Another approach: `template` means that `T` has a more specific *value* in the instantiation.
  - Consequence: `let U: Type = T` also works.
- [josh11b] Question: I'm writing an associated type in an interface, do I write: `let Element: Type;` or `let Element:! Type;`?
  - Associated types are written `let Element: Type;` -- it matters whether not whether the field is generic, but whether the value of the interface is generic or not
- [zygoloid] Suppose we have `T: Type` as a dynamic value.
  - `var x: T;` is invalid: `T` is not a generic constant
  - `var x: T*;` *could* be valid, because `T*` is "constant enough"
- [josh11b] We have said generic `T:! Type` does not result in multiple instantiations, specifically since the witness table is empty
- [zygoloid] It is directly observable in C++ when a function is instantiated, due to things like local struct types and local static variables
  - [josh11b] local static variables should be explicitly parameterized, and can't reference a parameter that isn't mentioned
  - [chandlerc] doesn't want to deviate from C++
  - [zygoloid] All the parameters look the same so they should behave the same
  - [chandlerc] the generic parameters are explicitly marked
  - [josh11b] template parameters more clearly trigger separate instantiation, I would like instantiation to not be observable for generic parameters
  - [chandlerc] want members of a generically parameterized type to have the same parameters as the type
  - [zygoloid] capturing local state; e.g. in Java when you have a local class it implicitly captures the this pointer from the enclosing class. Should Carbon have a local class capture the generic parameters from the enclosing context?
- [josh11b] do we have local statics at all?
  - [chandlerc] Match C++
  - [zygoloid] need a migration story for C++, including templates with local statics

        - [joshl] Concern is not wanting to make whether generics are instantiated visible
            - [chandlerc] Can use a table-based strategy for accessing local statics, even if the function is only instantiated once
        - [chandlerc] don't want to innovate here, local statics have cost but it matches C++
        - cost scales better than globals due to lazy initialization; might need to mandate that they have trivial destructors
    - [zygoloid] Think it is important that we have a mechanism to duplicate local entities in templates, think it might be important that we have a mechanism to *not* duplicate local entities in generics
    - parameterized type aliases like C++ template aliases, do we record the name and parameters of functions that return types?
    - [chandlerc] treat a type with generic parameters as a single type just as much as a function with generic parameters as a single function, so no problem if a function with generic parameters propagates those parameters to the type
    - [chandlerc] Re: dynamic types, somewhat unsatisfied unless we model the type erasure explicitly
    - back to template example and `let U: Type = T;` from before; two interpretations of templates:
        - interpretation 1: `template T:! Type` means that the type of `T` is something more specific than `Type` -- symmetric difference of the interface of constraints and that of the type; in above example, we only know `U` is a `Type`, not which type
        - interpretation 2: `template T:! Type` means we know more about the value; in above example, evaluating `U` as a generic constant would resolve it to the value of `T`, not to an opaque `Type`
    - [chandlerc] what kind better paves a rod for constraining this template with interfaces and potentially turning into a generic
        - [zygoloid] if `let U: Type = T` erases ("template affects type"), then it acts like a local generic
        - [chandlerc] Seems valuable to be able to write `let C: Comparable = T as Comparable;` to get a generic `C`. Can already write `x.(Comparable.Less)(...)`
        - [chandlerc] Like "template affects type" model because it makes this use case nice
    - What about non-type template parameters?
        - would like them not to have special set of rules
        - can we cast away the templateness?
        - Is there a difference between a non-type template parameter and a non-type generic parameter?
        - In addition to getting a more specific type, you get a template constant
            - This is partly interpretation 1, partly interpretation 2.

    - [chandlerc] We want `C` not to be a template constant, so need some rule like writing `:!` or `template` to indicate what kind of value instead of just getting it from the kind of the right hand value
        - `let template U:! Type = T;`
        - [zygoloid] No new ! since not triggering a new substitution? `let template U: Type = T;` to mean that the type of `U` is not `Type` but something more specific (which is what `template` does) but without triggering … something … to be monomorphized (which is what `:!` does).
    - [josh11b] Question: I write an associated type `interface Container { let Element: Type; }`, how do I write it in the `impl`? `struct IntStack { impl Container { ___ } }`:
        - `alias Element = Int;` // josh11b likes not having to restate the type of `Element`
        - `let Element = Int;` // Not a pattern (with the right meaning)
        - `let Element: Type = Int;` // Verbose
        - `let Element:! Type = Int;` // Unclear if ! is needed or meaningful
        - ?
        - [zygoloid] Don't like `alias` because an `alias` and a `let` seem like different kinds of things, and I'd like the `interface` and `impl` to agree.
            - Also: `alias Element = Int*;` wouldn't be a valid `alias` declaration if we want a name on the RHS.
        - `let Element: _ = Int;` // `_` meaning "inherit type from interface"?
- zygoloid will clone and take over the access control doc

2021-07-01

- Attendees: josh11b, jonmeow, chandlerc
- Rust still working out the semantics and requirements of unsafe code https://www.ralfj.de/blog/2019/01/12/rust-2019.html

- Attendees: josh11b, mconst chandlerc
- Thought about using whitespace sensitivity to distinguish between angle brackets and comparison
    - Turns out this is even hairier than distinguishing between different operators since it also could be a delimiter
- Syntax approach to generics (:!) is viable, simple, current default
- Question is whether the type-based approach is also viable
- Another alternative is the generic/dynamic bit is part of the name and not the type
    - so is not in the type parameters, avoiding `Vector(generic Int)`
    - if you have to override the bit, the keyword would go to the left of the `:` like where you put `addr`

- Different kinds of ways of using the type to select genericness without putting genericness all the way into the type system.
    - For example, this option: `name: T` would be generic if `T` is a type-of-type and dynamic otherwise, `generic name: T` would always declare `name` to be a generic parameter of type `T`
    - Concern: What if `T` was a template parameter and could be a type-of-type or not in different callers? Need some rule here
    - ChandlerC hesitation, when we use `T` to guess whether `name` is generic:
        - so it's in the type system? not really, but maybe this is confusing to people
- Well what if it was in the type system
    - What happens when you return a pair with a mix of generic and dynamic values?
    - Run the function once at compile time, once at runtime?
    - Force the return value to be all generic or all dynamic? Gives up ability to forward parameters to return value
    - Perfect forwarding almost always in C++ comes up in the context of passing parameters of one function to the arguments of another
    - Will forwarding come up for generic parameters? Don't currently have examples
    - May instead use reflection / metaprogramming to do forwarding
- What characteristics make for a good macro system compared to a bad one?

2021-06-29

- Attendees: chandlerc, jonmeow, josh11b
- Operator tokens #601 out of draft, ready for review
- chandlerc@ has liked using the Octotree Chrome extension for GitHub reviews, but the new "conversation" drop down seems to filter out resolved comments (nicer). Doesn't include pending comments though.
- Re: https://github.com/carbon-language/carbon-lang/pull/561/files#r659617342 do we want to support interfaces as base types long term?
    - Very similar, but slightly different performance characteristics
    - For C++ interop,
    - Long term, do want to support using an interface as an abstract base class in the type hierarchy. In the absence of multiple inheritance, we can make this efficient, even with C++ compatibility
    - Possible restriction, to improve the efficiency of the multiple inheritance case, could require you to use `DynPtr` for any parent other than the first, instead of allowing you to form "pointers to base type" which would require a different value for `this`
    - Including multiple vtable pointers in the object
    - Maybe have an opt-in to C++-compatible ABI on the Carbon side
        - Basically, a layer on top of / around `DynPtr(MyInterface)` that lets us create an object to explicitly replicate the C++ vtable/vptr ABI desired. Can maybe try to optimize this if our toolchain can

change the C++ ABI but gives us a hook for where we can build out compatibility when needed.

- Scripting deleting git branches

2021-06-28

- Attendees: josh11b, jonmeow, chandlerc, zygoloid
- chandlerc@ knows of some cases in the overlap between ABCs and polymorphic types
- zygoloid@ asks about cases which might be done with C++ private inheritance (but sometimes are done with public inheritance for convenience): inheriting from something like `vector` which was not intended to be extended
    - Concern around virtual destructors and slicing. Could potentially prevent unsafe conversions in this case.
    - Is this a case we should support? Or just document as not being supported?
- chandlerc@: even more overlap between object types and ABCs / polymorphic types; the latter two do encapsulation
    - Three roots: data types, object types, and mixins
- String types use inheritance as an implementation detail for e.g. functions that don't depend on template arguments
- LLVM's small string inherits from small vector; has a subtyping relationship, but no dynamic dispatch
    - Small string vs. small vector: maybe a good use for Carbon's adapter, it extends the interface; has no additional runtime state
- There are other cases of type hierarchy without any need to override base functions in derived types; has come up on multiple occasions
- C++ access control & encapsulation vs. other languages
    - other languages use more of a file or package scope, see https://en.wikipedia.org/wiki/Access_modifiers
    - Rust & Haskell: based on what is exported; if you don't export a way of naming the members of a struct then you can't
    - josh11b: would like tests to have an easy option to access internal implementation details
    - chandlerc: C++ isn't too bad here, maybe could be more convenient
    - zygoloid: maybe test code in the same library can declare itself a friend, so we can keep the public interface cleaner
    - Implication of the Haskell approach: you can always see everything in the same file, outside of the file it is only what is exported/public
    - Does "protected" fit with access control generally?
        - Instead: what interface does the class expose to different clients

```
// Before:
class X {
 public:
  fn A() { ... calls B() ... }
```

```
 protected:
  virtual fn B() { ... calls C(); ... }
  fn C() { ... }
}
// After:
class X {
  public:
   has a Y and creates a Z
   initializes Y with Z
   fn A() { ... calls Y.B() ... }
}
class Y {
  has a Z
  // Can't call A, which would cause an infinite loop
  public virtual fn B() { calls Z.C() ... }
}
class Z {
  // all stuff that can be used by Y
  public fn C() { ... }
}
```

    - Do we want a given class has a dozen different functions, and rules which say under what circumstances those functions can be use; or do we want to say the class has n different interfaces, and we have some rules that say which of those interfaces get exposed to which clients
        - 2nd thing that the first thing doesn't: doesn't depend on where the code is; refactoring is not going to change the meaning; can delegate
        - C++ language rule bugs mean that friend is weird
    - People write traits in the type system based on whether methods/constructors/destructors are private or public
        - don't allow delegation of permission
    - Should instead intentionally model this using the main typesystem tools
    - Interface is determined by the type, and you can have multiple different facet types for the same representation? How does that work out? We end up with the type inside the type definition `Self` is different from the public view of the type.
    - Inside an interface there is no way to name that distinction
    - Do we need access control for the casts?
    - Do we use zero-sized access control tokens to decide which functions can be called?
    - Do we need access control for types themselves? Is that enough to avoid the need for access control tokens / lock and key pattern?
    - If the friend has the type, it can do whatever it wants with it.
    - Model does allow delegation, which is desirable
    - How do we name the type, given that we are getting the restrictions by providing a way to name the type

    - Bootstrapping problem giving out keys/capabilities
    - Easy to accidentally expose private data by writing the wrong return type:

```
class X {
  // OK.
  fn Get[addr me: Self*]() -> X* { return me; }
  // Probably bad: exposes all private members.
  fn GetPrivate[addr me: Self*]() -> Self* { return me; }
  // ...
}
```

    - Is it valuable to provide "smaller scope private"?
    - Maybe `Self` is just not an exported name, available within the same library
    - Could catch issues of returning values that have unexported types
    - Have two things:
        - Want something enforcing encapsulation to catch mistakes
        - Want something that keeps things library internal
        - Don't want a third thing
        - Can we make them the same thing? Seems hard
    - Seen librarians need something finer grained than at the module level
    - Python gets away with very little
        - zygoloid: does have a problem with separate namespaces for base and derived types
    - Maybe don't need to be that inventive here, better to meet C++ programmers where they are
        - Still would like to fix delegation, "can I ask this" modeling in the type system, leads to friendship being broken in C++
        - No need to change granularity
        - Do we want to model library-level access control? Is encapsulating a module from other modules the same as encapsulating types from other types?
    - Physical encapsulation is different since it determines separate compilation
    - Do we end up with three things?
        - physical encapsulation of libraries (stuff in the implementation file and not the api file is not available outside of the library)
        - exposing multiple interfaces for a single type, providing logical encapsulation of a type
        - selectively exposing type interfaces to some consumers
    - Need some way to expose things differently to different consumers
    - C++ friendship can violate layering
    - Would it be fine if both sides opt into the friendship
        - a problem in C++ since generic code might know about an interface you expose
    - Forward declared "key" type in API file, can define the key and the function returning access in the test

    - If no need for long-distance (outside the library) friendship, all we need is the key type pattern
    - Maybe build system could say this library is private to a package, and could hold keys for package-private
    - What about protected? Another problem
    - For data types, the `Self` is the same as the public type
    - Maybe: the `Self` type of types generally is a `struct` type
- What are the use cases we want to support?
    - chandlerc: liked the breakdown
    - would like non-polymorphic single inheritance added: types where the subtyping hierarchy doesn't require dynamic dispatch; still is-a not has-a; LLVM data structure design; comes up as an implementation detail a lot: iterator derived from const_iterator; no slicing since not using pointers (or no adding data); different from private inheritance;
    - lib c++'s implementation of red-black trees, has three different classes for nodes and node-like things, some tree traversals only care about left-right not up, some care about up and not data, some care about all, use subtyping to avoid extra code for different node types where the algorithm doesn't care about the difference; can properly type all the nodes in the tree so the root of the tree is a sentinel; doubly linked list use the same inheritance trick to store different data for the sentinel node
    - reversible type erasure; want to have a bunch of things cast to some common type and later going to recover the original type from code that knows what type it is; in C++ can cast between base & derived but not between subobject and complete object
- See a separation between `struct` and other things
- Question: do we want to combine all data types into anonymous struct types?
    - gives up some type safety for data types, since it uses structural type equality
    - LLVM IR had these, worked great until you tried to link large applications and needed to deduplicate, led to N^2; particularly hard if they are mutually recursive
- Have justification for having named structs in addition anonymous structs
    - named structs allows recursion, have names
- Access to the Self type fits the initialization story
- Question: is `{}` for structs ambiguous with `{}` for compound statements?
    - `{` at the start of a statement always opens a block
    - Don't want to support destructuring putting into an existing variable to look like this:

```
var x: Int;
var y: Int;
{.key = x, .value = y} = map.find(k);
```

    - Destructing always in a pattern context, which is never at the start of a statement
    - Maybe allow `let {.key -> x, .value -> y} = map.find(k);`

    - Could also push back on {...} as a bare statement, using `with { ... }` when needed instead
- Mandatory braces where C++ has optional braces
    - consistency; a very simple rule
    - reclaiming braces for use in e.g. struct literals
    - being able to use if/else without braces in the expression context
    - blame "goto fail"
    - avoid arguments about when braces should be included: what if I add a comment? What if I rename a variable and it no longer fits on a single line?
- Digression into conditional expressions
    - `if <condition> then <value> else <value>`
    - Require parens around? Or terminating `fi`?
    - Reject anything ambiguous, no precedence with itself
    - Do we need to write: `(if condition then value else (value))` to avoid ambiguity? `(if condition then value else (value + 1)) * 2`
    - rule: words lower precedence than symbols; or words are lower or higher precedence than symbols, but not intermixed
- Square brackets -> homogeneous sequence literals
    - can't make a sequence of literals if they are singleton types
    - common type of a collection of values
-

2021-06-22

- Attendees: josh11b, jonmeow, chandlerc, wolff, gribozavr, zygoloid
- Old PRs
    - Auto-close old ones, but free to reopen ones that are still relevant
    - Lots of old principle PRs are still "draft", need to decide which ones should be reviewed, but maybe wait so as not to delay other work on e.g. generics
- Comments incoming on Values, variables, pointers, and references
- Design choices around name lookup #424
    - Use cases?
        - Shadowing to implement the use cases addressed by flow sensitive typing?
        - Unwrapping optionals or results, as in austern's comment
        - Single function interfaces?
        - Local vs. global? jsiek's argument that we should not distinguish is compelling
    - Namespace rules should match the member function rules; type name is in scope even for out-of-line method definitions even though that scope has been exited

        - Generally agreement with zygoloid's last comment
    - Generic syntax to replace provisional $s #565
        - Ways of saying something is constant?
        - Ways of saying something is mutable? Just `var`, including in parameter lists
            - `var` parameters can affect the caller, maybe: caller must pass a temporary
            - overloading on `var`: reusing a `Tensor` buffer if the argument is a temporary
            - is there a way to have `var` behavior with temporary parameters, but copy if the parameter is not temporary
            - maybe `var` means this last thing and `take` for only matching a temporary
    - Method syntax #494
        - Trying out syntax #2, not committed to it yet
        - Prefer the `addr` approach to implicitly taking the address if the type is a pointer
        - Less need for overloading on `addr` if we aren't returning references
        - Interop with C++ classes that overload on `const` and return a reference or const reference is going to be a bit ugly
    - Overloading?
        - Best match vs. first match

```
overload fn F {
  (Int) -> Int;
  (String) -> String
}
```

        - Josh likes the rule that you can't define a new overload for a function after calling it, to avoid the definition of the function changing after use
        - Carbon calling C++ is going to have different rules than Carbon calling Carbon. For example, open overloading: we see in practice that some C++ build targets have different overload sets for the same function as other build targets.
        - Chandler originally preferred best match, but likes aligning with pattern matching:
            - first match wins
            - error if any pattern is unmatchable, since subsumed by any earlier pattern
        - Concern about expressivity differences:
            - best match doesn't have a good way of resolving ties
            - first match has difficulty with "earlier pattern could match but would prefer later more-specific match". Example: String vs. StringView, can convert between the two, so will always match whichever is written first.
                - Need a way to say "only match `String` exactly, otherwise go on to the `StringView` match"
    - Interface implementation ("impl") syntax #575

    - chandlerc@, zygoloid@ have very mild preference for including `as` in inline impl
    - `external`?
        - helps if it is hard to tell if the `impl` is inside the struct definition
        - will probably call these "external impls" whether or not we use the keyword
        - won't allow external impl inside struct definitions, or interface definitions
        - low stakes, easy to change either way
        - more likely to notice if its wrong if we include the `external`
        - is ambiguity resolved by looking at whether it is defined in column 0?

2021-06-21

- Attendees: josh11b, mconst, chandlerc
- #565: generic syntax
- structural interface stuff:

```
interface Bar {
  method (me: Self) F();
}
structural interface Foo {
  impl Bar;
  alias G = Bar.F;
}

struct Song {
  impl Foo {
    method (me: Self) G() { ... }
  }
}
```

- #494: method syntax, particularly things like addr me: Self* are converging
    - Don't want automatic conversion to pointer without marking in the declaration
    - Keyword not part of the type so it does not bifurcate the type system
    - Interaction with overloading?
    - All names in the overload set would require `addr` if one does
    - A method that has a constraint, can express that in the type of `me`
    - Equivalent of C++'s overloading based on const-ness in Carbon?
    - No C++ const pointers binding to Carbon let values? Or only in unsafe code?
    - Tension between preserving semantic information associated with pointers and deduplicating redundant code

    - Can avoid allocations or copies if we have overloads on whether the argument is a temporary:

```
// 1st overload:
fn TensorAdd(x: Tensor, y: Tensor) -> Tensor { ... }
// Would prefer these second overload if it applies,
// since it could reuse the buffer of x
fn TensorAdd(move x: Tensor, y: Tensor) -> Tensor { ... }
var (x, y, z): (Tensor, Tensor, Tensor) = ...
var r : Tensor = TensorAdd(TensorAdd(x, y), z);
// inner TensorAdd uses first overload
// outer TensorAdd uses second overload
// Could also have one or two more overloads so we
// can also reuse the buffer of y.
fn TensorAdd(x: Tensor, move y: Tensor) -> Tensor { ... }
// Would need a fourth overload to resolve ambiguity.
```

        - Main concern with having different overloads where some use `addr` is how to prioritize between overloads? With `move` above it is clear that you want to prefer temporaries with `move` parameters. Do we want to prefer lvalues with `addr` parameters?
        - Probably just preventing these overloads until we have use cases and better understanding on how to prioritize between overloads.
        - Maybe: manual prioritization using the order that overloads are declared?
        - Don't allow new overloads for `Foo` after the first call to `Foo`. Code will generally just declare all `Foo` overloads before defining any of them.
- Discussion about forward declarations vs. name lookup can see names later in the file
- Marking dynamic type parameters?
    - dynamic `T: Type` is actually not expensive, since there is no witness table
    - Might make a lot more sense for dynamically (type-erased) provided types, with dynamic witness tables (or equivalent) to still be a compile time (generic) type, but one that type erases the call-site type
    - Fully dynamic types might be something much closer to what we get from `typeinfo`, maybe allowing something like this?
        - `fn Lookup[T: Type](t: PtrTable(T), key: String) -> T*;`
        - Actually this is likely closer to `T:! Type` rather than having constraints
            - this is just `void*` but with type
        - more `typeinfo`-like dynamic example is probably better
    -
    -
- Maybe there is a `TypeId` interface that all types implement, but you have to opt in to the runtime or code space cost for it

- Is most code non-generic? Generic code is definitely not rare. But hard to optimize builds where generics are pervasive, and we've tried a lot in the similar situation of link-time optimizations.

```
fn Lookup[T: Type](t: PtrTable(T), key: String) -> T*;

C: Dyn(Container)
// Chandler: [DynC:! Dyn(Container)]
// Josh: [dynamic DynC: Container]

[T:! Type]
[N:! Int](arr: Array(Int, N))

fn F(... bunch of args ..., result: Reduction) -> result;
fn F(... bunch of args ..., r: Reducer);

fn Sort(..., comp: CompareOperator);

// container1 and container2 have the same type, but it is not statically known
fn CompareTwoContainersOfTheSameType(container1: ?*, container2: ?*) -> Bool;
// Josh world:
fn CompareTwoContainersOfTheSameType[dynamic C: Container](container1: C*,
container2: C*) -> Bool;

fn CallsTheAbove[dynamic C: Container](container1: C*, container2: C*) -> Bool {
  return CompareTwoContainersOfTheSameType(container1, container2);
}

// This, but I want the element type of the two containers to be the same, but not known
statically
fn CompareTwoContainersOfDynamicEltType(container1: DynPtr(Container & ?),
container2: DynPtr(Container & ?)) -> Bool;
// Josh world:
fn CompareTwoContainersOfDynamicEltType[dynamic T: Comparable](container1:
DynPtr(Container & {.Elt = T}), container2: DynPtr(Container & {.Elt = T})) -> Bool;

// Fine
fn CompareTwoContainersOfStaticEltType[T:! Comparable](container1: DynPtr(Container
& {.Elt = T}), container2: DynPtr(Container & {.Elt = T})) -> Bool;

// Before:
fn F[T:! Foo](p: T*)
// J: fn F(p: DynPtr(Foo))
// R: fn F(p: dyn Foo*)
// Alternate before, Rust only:
fn F(p: impl Foo*)
```

```
// In Carbon
fn F(p: (_:! Foo)*)
// Swift is considering for static case:
fn F(p: some Foo*)
// Swift dynamic case is:
fn F(p: any Foo*)
```

2021-06-15

- Attendees: josh11b, wolff, chandlerc, gribozavr
- Josh mostly on vacation, needs to do more work on generics overview doc
- Value patterns as function parameters #578
    - Leads don't want `F(Int)`'s current meaning, need some fix
-

2021-06-14

- Attendees: josh11b, zygoloid, chandlerc
- Function and variable bindings, method syntax?
- Do we want to control e.g. let vs. generic on a binding or pattern level?
- Angle brackets would be suggestive of generics to users. Would we write something like: let <v>: (Int, Int) = ...;?
- Not excited about triangle brackets <|...|>
- Can drop square brackets if there are only angle brackets inside?
- Square brackets instead of angle brackets --> means both deduced and generic, but means no way to say deduced non-generic; which is something we do want to be able to say (at least for the type case)
- Dependent typing? Interesting for passing in an allocator without storing it in the object

```
struct Node(dynamic ctx: Context*) { ... }

struct Context {
  fn MakeNode[addr me: Self*]() -> Node(me)* {
    return new Node(me){ ... };
  }
}

fn DoStuff[ctx: Context*](p: Node(ctx)*, q: Node(ctx)*) { ... }
```

- For types, need to say template, generic, or dynamic for types explicitly
- If "generic" is spelled "name :! type", then maybe template is spelled "template name :! type"
- Somewhat back to function parameters and variables -- how different are they?
    - chandlerc feels maybe more similar than expected even? not clear
- What information should be in the typesystem (taking off from #typesystem question from zygoloid)
    - lots of pondering about whether "everything" could work -- maybe?
    - does this mean value category? in a sense yes, but maybe not as we know them in C++

2021-06-10

- Attendees: zygoloid, josh11b, dabrahams, gribozavr, geoffromer, jsiek
- [zygoloid, josh11b] Some discussion about whether receivers can be smart pointer types
    - Might motivate different syntactic choices for methods, but it's unclear that it's compelling to allow (for example) a method to expect a unique / shared pointer to its object
- [dabrahams, jsiek, zygoloid, geoffromer] Swift interpreter deep dive, 2nd session
    - [dabrahams] I have written design notes and implementation notes. The important things for the interpreter are AST.swift, Memory.swift, Type.swift, Value.swift, and Interpreter.swift. We haven't looked at Memory, Value, Interpreter. Let's talk about the memory model. I knew that carbon is going to care about things like user-defined initialization, how many times things got copied, which are important for performance, so I tried to establish a memory model that allowed us to count those things. If you look at the toy interpreter, it didn't try to model memory. It passed values around between different parts of the system. Here we have values that live in memory. It is fine for the system to optimize some of that away, put it in registers or whatever. But we have a good foundation that allows us to model memory.
    - It is also important to represent sub-parts of objects. I went through a couple of schemes for representing it. An address is an allocation, which is just a counter, every time you allocate you get a new allocation number, plus some keypath to the specific subpart of the object that you're referencing. All objects that the interpreter manipulates conform to this protocol. A keypath is a list of subscripts and property accesses. The implementation notes say that these paths can be opaque to work with, and that's why there is a string description that provides a human readable explanation of a key path. When we're addressing subobjects we are addressing things that are addressed like a tuple with an integer, or by a property name, and stored in that address is some value. That makes most subobject accesses quite simple. But if you look at for example function type, it does not store a value. It stores a tuple type and a type. Both of them conform to value, but you can't just use the trivial way of

composing keypaths to get at the subparts. Possibly for function types we wouldn't care, but in order to avoid introducing copies that we can avoid, we should be able to construct things in place: allocate memory, fill it in with some provisional values, and then model initialization. That's why there are two different ways of forming addresses. One is with the operator that I overloaded (dot caret), that's used for integer positional field access and named field access. The other one is an explicit construction of an address, that allows you to get to a part of an object. Tuple type is not a value, but you can arrange a key path that lets you get there. The hoops involved in arranging the kepath is why you want to consider representing a keypath as an array.

- [geoffromer] Where is keppath defined?
- [dabrahams] it is built into swift.
- [dabrahams] FieldID is this enum with two cases, either an identifier or an int, most of the subobjects are referred to in one of those two ways. This dot caret operator composess a new address with the extra dereference, either via subscript or name. There are some convenience overloads that allow you to build a FieldID if you have an identifier or an int. I thought it is important to make it read naturally in the interpreter, but I won't be insulted if anyone takes out the overloaded operator.
- Let's look at values. The reason for TypeID is to capture a type and make it conform to Hashable. There are ways to represent a type as a runtime value, but it is not hashable, and TypeID works around that limitation. If we want to look things up in a dictionary based on the type… Wait, it does not have to do with hashability, it helps with downcasts.
- FieldAccess abstracts out the subscript operator that you can apply to any value or tuple of pattern, tuple of pattern is obviously syntax, not a value. And these extensions provide you convenience similarr to the dot hat ooperator.
- Value is not derived from FieldAccess because of other Swift limitation. FieldAccess has an associated type, and if Value was derived from it, it would not be able to be used as an existential. So Value can pressent you the dynamic type of the value. It also has a subscript requirement, which overlaps with FieldAccess rrequirements.
- Next, some subtle side-effects of using key paths. This mahchinery is needed to turn a thing that is not a type value intoo something that is, so that we can access it with an address.
- Next two refined prootocls, AtoomicValue and CoompooundValue thatt bring together everything that we sa w previoously. AtomicValues have a default implemenntation of subsccript that sayss that it has no subparts. CoompooundValue brings them together and expects you to define yoour subscript.
- Some specific valuess. FunctionValue is an atomic value that contains the AST node for the function definniton, and the dynamic type of the function.

- [jsiek] Compparing the value we usse in the innterpreter, comparing with the AST where you use sum types, here you use a protocol-oriented approach. Is there a reason?
- [dabrahams] There is a reason, but it might not hold water. IntValue is an Int. Maybe we don't want to doo that. Because now Int has a subscript operator. Maybe you want to make IntValue a struct that wraps an Int. This is one reason, but I don't have a position against using sum typpes for values.
- [jsiek] I was thinkig about having interop between C++ interopreter and the C++ code? Then you need the memory layout to match.
- [dabrahams] My sense is that if you were to do that, youo would be serializing/deserializing at the booundary of the interpreter. Tryying to integrate that other information might be too harrd.
- ChoiceValue is a thing with parts. The discriminator is accessible only to the interpreter. The payload is a value. I used the didSet fetarue, it runs after you set the value, and it checks an invariant.
- I'm using dynamic_type_ because I want to preserve the type iformation thatt I'm referring to a Choice. I also present it as dynamic_type here.
- The initializer is copying values except for one thing. I rewrite the paylooad to trigger the didSet.
- StrructValue is similar but simpler, theer is no disciminator.
- Alternative values fall out from the ccurrent design of the language. I think because a cchoice might be created from a value without parens we have to represent a value that might be a choice or a value on its own.
- Type is both a sum type, but also has a subssccript. So types are hybrid. It might be better to use sum types.
- The model for memory is fairly script, allowing to doo type-based-alias analysis and what not. You may loosen it of course. Memory has a dictionary that maps the allocation number to the top level value. When you allocate memory, you need to give it the type that you will store in it, and whether the thing is mutable or not. Every bit of memory, including stack, gets allocated in this model. When you allocate memory, it is filled with uninitialized value. It is either of this special value type called uninitialized value, or it is a special struct instance called UninittaailezedType. I have imbued unitialized values with static type informaation. For example, a function type has a tuple type in it. It is not just a value. If we want to represent an uninitialized function, we need something to put into the tuple.
- [jsiek] Types are subsets of value. If you want something uninitialized that is in the set of types, you need an uninitialized type.
- [dabrahams] In principle, I could use just the uninitalized type which is also a value. But its type is a type, so that contradicts our type system. There may be a more elegant way to represent partwise initalization.
- alloocate(): allocate a memory location, we set up the loccation with an uninit value, we record if it is mutable, create an address and rreturn it. initialize() is a bunch of checks to make sure we're not abusing memory.

Then it stores the value into the addrress. here it uses the key ppath to get at the subpart where the value is to be stored. There is a special key path that refers to the whole object.

- deinitialize() takes an address and makes it uninitialized.
- deallocate() is mostly sanity chcecks.
- In assign() nothing is being done to break out assignnment of subparts. That would actually have ann impact on the interrprreter.
- [jsiek] Say we have an assignment in source code. LHS refers to a subobject, field access. How does the interpreterr get to here, is it using assign() of the whole object?
- [dabrahams] assign() ends up being used on a subpart. The address is computed using the dot-hat operator, and then assign() is called.
- [jsiek] so the sourrce annd target in assign() can rrefer to suboobjects?
- [dabrahams] yes. But once you model assignment of oobjects with user-defined assignment semantics, you want to model assignmnt as a bunch oof smaller steps.
- [zygolooid] Is there a reason why assign() takes a source address but not the source value?
- [dabrahams] all values get put into memory. So you can retriveve the value from there.
- [jsiek] There's perhaps an API design decision. not wanting peoople to mess arouond with values, only work with memory, addresses.
- [dabrahams] I think that's part oof it. Once you start enabling the interpreter to handle values directly, you can easily accidentally do it, and that's not supposed to happen in a proogarm.
- Next is Interpreter.swift. The interpreter is using trampolined continuiation-passing stsyle, it makes functions a significant part of the dessign. It is important to push the interpreterr one stsep at a time, forr example, to model threads. We doon't have coroutines yet.
- The type Onward is cutely named due to how it is used inthe code, but effectively it is a C++ function object, that is a wrapper over a Swift closure. So Onward is like a std::function. It wraps one of these Next functions.
- The Conssumer is like Next, but it takes an extra argument at the beginning. It is useful for expression computation.
- I overloaded ooperator fat arrow. It becomes clearer when you see it being used.
- "return a => proceed". I want pass "a" into proceed, but I'm not invoking it, I'm making it my continuation. All functions here take a Consumer<T>, or Next, and return Onward, which is the rest of the computation. Every function should do some nominal atom of work and should pass the rest to the system as some Onward.
- CallFrame is everything that needs to be saved by the interpreter when you call a function. A set of persistent allocations, that will last to the end of the scope. The oother things are called ephemeral. We're storing the expression only forr diagonstic purposes. If you're wondering what we're

stooring, you can get to the expression and find what is left behind. Then there is 'local' which maps simple bindings, variable name, colon, type name, too the address where they are stored. Yuo never allocate a local explicitly. You alloocate the thing on the RHS and then you pattern match to it, and it bindss the local variables too addresses in the RHS. That's noot neccessarily the semanticss you want. Some parts might stick around loonger, but we have less copying. Function calls writ etheir results intoo a caller-supplied address. Then we have onReturn, onBreak, onContiue, similar to what we had in the tooy example. We set up the onReturn continuiation at the beginning of the function in the case when someone calls return.

- [jsiek] Hoow does this data layout handle nested loops?
- [dabrahams] I formalized the idea of saving the call frame, I didn't do it for the loop context, but it is saved.
- [jsiek] Due to CPS, we can use local variables to save these continuations.
- [dabrahams] What's in the interpreter? The program. Call frame. A separate dictionary to map globals to their addresses. Memory, which we saw already. Next step to be executed. A bool that tells us if we're still running. An error log in case a few things pile up, it is going to stop execution, but we can put warnings there too. There is tracing, that stores a trace level, which corresponds to indentation. And here is a property map. I sometimes reach into the program directly, but for types I reach so often that I made a computed property.
- The constructor for the interpreter creates the initial call frame, sets up onReturn, which cleans up all allocations in this call frame and calls terminate.
- The next step executes the body of main. Capture lists make it a bit confusing which code is delayed, and what is executed now. Capture lists are inside curlies which are delayed, but captures are executed eagerly.
- I have a separate step() function that may seem like a trivial wrapper but I want to ensure that we have the ability to go one step at a time.
- When you're type checking you don't have an executable program. In this next init we pass the expression, program, the AST, and results of name resolution. It creates an executable program. Then sets up a call frame to evaluate it.
- [geoffromer] Type checking is still in progress. Do we worry about type information for interpretation being unavailable?
- [dabrahams] I was able to write this by passing the type checker by value, so the Swift type system provides us a guarantee.
- [geoffromer] You don't branch on whether the type has been completed yet?
- [dabrahams] no
- [jsiek] What if a buggy program referred to a runtime value from a type context?
- [dabrahams] I didn't check for it because the c++ implementation didn't do it. But the type checker could check for it.

- [jsiek] in the c++ version we had two symbol tables, one for compile time things and runtime things. You'd try to look up a runtime value in the compile time table and it wouldn't be there.
- [dabrahams] tracing utilities got increasingly more complicated as I wanted to capture source locations of the program being interpreted as well as the location in the interpreter.
- The whole idea of using mutable projection and inversion of control and swift's modify access was very interesting to chandler, as a way to avoid creating a reference count.
- run(). At every statement the ephemeral allocations from the previous statement should have been cleaned up, we check that. These look pretty straightforward except for dealing with ephemerals. Expression statement evaluates the expression. Here we don't pass in an address, and it creates an ephemeral. Sometimes it is important, for example you need to evaluate a function call directly into where its result should go. The dynamic nature of this ephemeral tracking should translate into something static. But it is worth convincing yourself of that again, probably. Part of the idea is to end lifetimes as soon as possible.
- Maybe the project should settle on some kind of suffix for expression, type, address. I usually don't encode these kinds of information in a name, but I have never been in a domain where I have so many aspects of the same thing to talk about.
- Initialization is a pattern match. First we allocate the persistent storage for RHS, we will bind variables to it, and we don't want to copy. This allocate() is a method of the interpreter, not memory, it uses memory, but it has the form of CPS, to ensure that we are not putting together things that should be separate steps. Maybe allocation does not need to be a separate step because it will be done statically in a real system. After we evaluate, we match. If it does not match, we have an error. We want to ensure the absence of errors in the type checker, but it is not done yet.
- There are cases when you want to calculate a value and use it in the interpreter, like the condition of the if statement. This is what evaluateAndConsume() does.
- [jsiek] are persistent allocations tied to locals?
- [dabrahams] yes, but i'm not sure. The function call expression allows you to use ephemerals. But I can't call it locals, because each allocation does not represent a local, it is a RHS of the initialization.
- [josh11b] I'm surprised to not see inScope() in more places, like in control flow constructs "if" and "while".
- [dabrahams] That should be fixed.
- [geoffromer] I'd suggest we discuss design differences from the C++ implementation.
- [dabrahams] I should describe the rationale for using key paths. I think it is important for humans to handle an entire value on its own. That creates a need to have a representation of a value that contains all its bits, and an address that points into it.

- The CPS is the main tthing that is different from the C++ implementation. I ended up here moosly because once i tried replicanig the c++ implementaion, I found myself making arbitrary choices when organizing the code. When I chhanged to CPS, the thing wrote itself. When I was making an explicit stack, and deciding what things go there, I felt like tthere was no principle behind how to shape the code. I'm not saying it is an ideal organization, I don't love the pyramid of doom.
- I tried to make sure that everything that could hide any user-written code got its own step.
- [jsiek] Yes, you need at least that level of granularity.
- [dabrahams] What I found is that it is not always possible to ensure that you doo real work in every step.
- [geoffromer] What makes me nervous about CPS, it seems like a lot of the information thatt's reresented in data nsstructures in the C++ impl right now, is represented in local variables and captures. It makes me nervous that it is less accessible to the code.
- [dabrahams] I have been thinknig about that a lot. I was unsure about it, I think I could reassure you, if you think about what Jeremy calls PL101 interpreter, 311 interpreter, the standard recursive descent sstraightforward interpreter that does not get you separate steps. If you think about that as a natural thing, all of the same things that are stored in local variables here in Swift, would be local variables there. I understand wanting everything to be in data structures. But I don't think anyone is worried about 311 interpreters being hard to understand. If you need to look at these things in te debugger, CPS changes what you see in the call stack. Moore htan that, when intnerpreting some of the programs, it is hard to step into the next funnctioon. Their handlingn oof swift isn not perfect. You might need to step by instruction until it lands in the function you want. But that's not CPS's fault.
- [jsiek] Foor a lot of the interpreter, the difference is not going to matter that muchh, but when you're implementig tricky contrrol features, like delimited continuations, I was looking at the data, I was priinting the data. IS there a way to see this data in CPS?
- [dabrahams] You can printn, trace, anything. The trace mechanism is pretty good att being able to corellate swift code and carbon code. What might be weird if you need to look back sseveral frames back, if you want to print basically history, local variables from the earlier call frame. If you need to see thtem, you need to use a debugger.
- [jsiek] I'm more interested in inspecting future, the continutation. Ensure it iss the right thing.
- [dabrahams] If you're in a debugger, lldb will tell you which closure it is, but the inside is oopaque. If youo're not in the debugger, good luck.
- [jsiek] It will point you to the swift functtion. What you're more interestsed is the carbon code.
- [dabrahams] Youo could come up with explicit representation for Onward that included some payload to describe tthe Carrbon code. It will be less

    - wieldy, but make the thing more visible. You can put an Optional<Any> into Onward and put extra info into it for debugging.
    - [jsiek] or the line nnumber in the carbon program.
    - [dabrahams] yes. OTOH, we would have the ssame opaqueness if we had rreal corounites. I have been pretty libeal in leaving extra infomation everywhere, and I think it is the more practical way to make the ssystem more debuggable.
    - [geoffromer] Any other differences in the Swift implementation?
    - [dabrahams] General differences, trying to represent as many distinctions in the static type system as possble, types vs values vs patterns. The attempt to do somethig that could model memory and user-defined five operations, that's very significant. There are comments. I think most of these things are obvious to you all.
    - [jsiek] I think you mentioned things that I forgot about.
    - [dabrahams] Another significant thing is that I only used safe constructs.
    - [jsiek] In the dispatching in the interpreter, it is dispatching based on the static type of the expression. Older versions were looking at dynamic types of values. What's nice about using the static type information, is that it is super clear that it is possible to implement it as direct calls in a compiler. Traditionally, people rely on their understanding of how the type checker works to convince themselves of this. We could also try to decide statically when the ephemerals are going to be deleted, by inserting these actions into the AST, then the interpreter would not need to make these decisions.
    - [jsiek] In the c++ version, the type checker produces a new expression. One way to deal with function call ambiguity, the type checker could disambiguate and produce different kinds of expressions. When Swift is not producing a new AST node it is avoiding all the extra junk needed for constructing new AST nodes. So there is a trade-off.
    - [dabrahams] I described this in my writeup, why we are not creating new AST nodes.
    -

2021-06-08

- Attendees: dabrahams, geoffromer, chandlerc, zygoloid, josh11b, jsiek, jonperkins, gribozavr
- Executable Semantics language choice
    - https://github.com/carbon-language/carbon-lang/issues/559
    - [darbrahams] gave a walkthrough of toy Swift interpreter
    - [geoffromer] gave a walkthrough of C++ version of a similarly structured interpreter, highlighted differences from the Swift version
    - [zygoloid] consider using the visit with an overload of lambdas approach?
    - [geoffromer] yes, not clearly better
    - [chandlerc] it can increase lambda nesting

- [chandlerc] even though we can make some incremental improvements to the readability of the C++ version, I think it represents the baseline level of readability of what the production C++ version will look like
- [zygoloid] How is the 'auto' in the lambda in the parameter of Evaluate() instantiated?
- [geoffromer] It is passed to Evaluate() that calls std::visit with this lambda. If you pass in a lambda that takes only one type, the PartialVisitor in Evaluate() will adapt it to a generic lambda, and will abort the program if anything but an object of the type you can handle is passed in.
- [josh11b] What do you all think about the tradeoffs between the approaches?
- [jsiek] what are we trying to decide? There is a big design space. The question of the language is only one dimension. Dave has been investigating more. I captured some of them below here. Most of these choices are independent of the language choice. Then there is a question, given a set of design choices, which language supports them better? So I wanted to highlight that what's going on is more complicated than just the language question. So I wanted to solicit opinions on other points, like whether we want continuation-passing trampolines. There is also destination-passing style. Questions about name lookup happening once ahead of time or in the interpreter. Which decisions are better in general, and which work better in Swift and in C++?
- [geoffromer] +1, some of those choices are even more interesting than the language question. So far in the leads question I tried to carve out just one dimension and not juggle all questions at once. Since different designs might work better in a certain language, it might make sense to consider the questions together. I think however that we can make the language decision separately. I think this toy interpreter is the best case for Swift, where it has the trailing closure advantage for the continuation passing style. My inclination is that C++ is a better choice for this project. That's why I think we can make this decision now. Maybe we should look at a language that has coroutines support. It looks like both Swift and C++ will be getting it in similar timeframes, so maybe we shouldn't worry.
- [zygoloid] I think it will be important to be clear on what our goals are for executable semantics. Clarity and readability are the most important. We want to be able to understand the meaning of Carbon programs without executing them. It should also be hackable, people should be able to easily implement a language extension. So the choice I see is less boilerplaty code on the Swift side, where we have less mechanism and more of the semantics. On the other side for C++ we have familiarity for our audience that knows c++ but not Swift. While listening to Dave's presentation, I haven't seen a part of the code I didn't understand. There is a question on the hackability question, do people need to actually learn Swift, or if they can get away with cargo culting? I think that will get most people far enough. I want to see the list of choices from Jeremy. I think

explicitly capturing values has improved clarity from the readability point of view.

- [josh11b] I felt like Dave's explanation was very useful to me to understand what's going on in the code. Some sort of bridge for C++ programmers to get from things who are not familiar with some Swift constructs that are a bit maybe magical, is important to produce. I feel like I eventually understood where it was going.
- [gribozavr] Audience is language developers/geeks/early adopters likely to have a broader language background than typical C++ programmers, people contributing to executable semantics will probably be language geeks familiar with many programming languages. I don't think that Swift, which tries to be a C-family language, will be a problem. C++ used more than the narrow subset commonly used in application code.
- [dabrahams] A number of advanced constructs were involved in the C++ code.
- [zygoloid] +1, I think the sum types in Swift brought a lot of clarity. They also helped decompose the code better than in C++.
- [geoffroomer] +1 that target audience will be language geeks and early adopters, so I think you're right it wouldn't be too big a lift to ask them to pick up a new language. OTOH, to pick up a new language even as close as Swift, even if it won't take too much time to understand enough to read it, there will be more work to become proficient enough to contribute. Also a question of bringing new people. Also what happens right now is a question. Dave is leaving the project, I'm very much a novice in Swift, and the only other person familiar with Swift is Dmitri, and he dooes not have much time to spend on the executable semantics. We might end up with executable semantics written by Swift novices.
- [gribozavr] Rust would be fairly similar for Swift in this application. zygoloid knows Rust.
- [jsiek] With Dave on the team I felt very comfortable in being able to lean on him in becoming familiar with Swift. I'm comfortable, but not fully proficient, so I feel a bit uneasy.
- [dabrahams] I probably have been writing Swift longer than anyone in the world, so you shouldn't compare yourself to me. You should look at how people adopt and learn Swift in the real world and what kind of resources are available to them.
- [josh11b] Richard on Rust?
- [zygoloid] I have relatively more experience in Rust, but I wouldn't consider myself a Rust expert. I think if we pick Swift or Rust, there would be some cost. It will be lower for me for Rust.
- [gribozavr] executable semantics is just an application, we're not hacking on the Rust compiler. no need to be an *expert* on the implementation language to write an app.
- [dabrahams] Given that goals of carbon are close to goals of Swift and Rust, some characteristics they will have in common. For example, sum types, support for safety. There are certain inconveniences that you bump

into due to strong type checking, that you won't see in C++. Aside from making it harder to experiment with ideas and architectures, those C++ quirks become more of a problem than those safety speed bumps, so all you see is C++ quirks. So while I think familiarity counts, I think if we want to build a grreat language, you should be using and experiencing other languages to get ideas on how to achieve Carboon's goals. My perspective on programming vastly changed by my experience in Swift. Before that I was primarily a C++ programmer. I dabbled in Haskell, APL, but having to work with Swift exposed me to other, new ways of working with ideas.

- [chandlerc] I don't think what I will say will change the fundamental tradeoff, but. I had an interesting time reading the Swift code, and I understood how it works, everything that it did. And then reading the type checker, I didn't understand. Fortunately, Dave gave me a walkthrough. It would have taken me a large amount of time to credibly make a change to the type checker in Swift. So I think the familiarity cost will be a real cost.
- [zygoloid] To what extent was it due to the language, or due to the patterns?
- [chandlerc] When trying to understand the patterns, the unfamiliar language was in the way. When I did not know the conceptual model, I had to jump through language hoops. I could have gotten through it myself, but it is a speed bump. For example, I had trouble connecting the implementation of the AST nodes to how they are being used. Dave explained it to me, and then it became super obvious to me. I don't think I would have the same problem in C++. I was just looking in the wrong places for things I was interested in.
- [dabrahams] What you're saying sounds sensible to me, but I can't concretize it, itw would be useful to me to understand it concretely.
- [chandlerc] The interplay between TokenRange and Identified, and how code uses these facilities. It is not a bad design, but when you laid it out to me it was much easier to understand than if I was reading the code.
- [dabrahams] I understand and agree. Those are indeed subtle parts.
- [geoffrromer] I think Chandler's experience was very much like mine. It seems plausible that those sorts of problems can be fixed with documentation, particularly documentation written with a more C++-speaking audience in mind. This is a flip side of the concern that I raised earlier, Swift novices implementing executable semantics. Conversely, as we become more proficient in Swift, the curse of knowledge kicks in, and it would become harder for us to translate for others who don't have a Swift background.
- [dabrahams] Well, a different option is that maybe that code shouldn't use such a subtle idiom. You could also use classes and class identities would be more straightforward than using source locations to map it. When I wrote it, I thought I would be mutating the AST a lot, but it turned out that it is immutable.

- [richardsmith] I think re: the curse of knowledge, I think C++ really hits it, because it requires in-depth knowledge of C++. I suspect that for someone with only modest knowledge of C++, the Swift implementation might be more approachable than the C++ one. About the equality comparisons we could write them out explicitly and avoid relying on synthesized code relying on location equality.
- [chandlerc] I think in both C++ and Swift versions we would be successful in hiding the subtle parts behind abstractions. In C++ the abstractions might become harder to write, and will be bigger, but I believe they can be extracted. I think baseline familiarity is more interesting. I think this implementation approach decreases the risk. Someone coming in the first time wouldn't need too make changes to these subtle abstractions. While I think there is a cost associated with familiarity, I don't think it is huge. I think exposure to different programming paradigms is more important, and having syntax for bread and butter things that you do often, than constrained and familiar C++ syntax that creates boilerplate. I think that's the most important tradeoff.
- [zygoloid] There is going to be some chunk of executable semantics that is just machinery -- reading files from disk, producing diagnostics etc., that are not related to describing semantics. I think that we should be thinking of as code that we can maintain in any language. If we factor that out, I think the implementations are similar in complexity. I think the C++ version has syntactic overhead in terms of the sum types etc.
- [chandlerc] We should not overextend the horizon of this. Either the Carbon experiment will end because it is not worth doing, or we will succeed, and will want to rewrite executable semantics in Carbon. So I think we're choosing the programming language that we will use until that day arrives.
- [josh11b] Do we need interop between executable semantics and other code?
- [chandlerc] I think if we had it, there might be interesting uses for it,, but it does not seem a priority.
- [geoffromer] That sounds about right. The goals of executable semantics and production toolchain are different. So if anything we should be a little averse of code reuse.
- [josh11b] For example sharing tests?
- [zygoloid] Sharing tests as data will be fine. But also sharing diagnostic infra would be good. I would be wary of using the semantic ports of the compiler executable semantics.
- [chandlerc] As a lead I have enough info. Richard?
- [zygoloid] Geoffrey, as a person working on executable semantics, do you have concerns working on the Swift version? Any estimate of costs?
- [geoffromer] Concerns? Yes, I don't know Swift well enough to estimate the costs. There will be some time window where I will be slower in Swift than if I was working in Swift. At some point I might be faster in Swift. I

    - don't know where the crossover will be. I suspect it won't be too bad, I won't be deadlocked in Swift.
    - [zygoloid] Are the two implementations in feature parity?
    - [dabrahams] Nearly. I'm connecting the interpreter to the type checker. When that is done, they will be essentially the same. There are some known differences, some test programs are only accepted by one version. There are some things that the Swift version rejects because C++ version might be too permissive. You can look at passing/failing tests to compare the two.
- [jsiek] Cataloging Exec. Sem. Design Choices
    - swift vs. C++, language features used in interpreter
        - sum types vs. variants (as in Geoffrey's prototype) or unions or classes
        - trailing closure syntax vs. closure as regular argument
    - trampolined continuation passing style vs. explicit stack
        - identified continuations (e.g. onReturn) versus inspecting the stack
    - destination address passing style vs. returning an address
    - name-lookup as first pass prior to type checking
    - deallocation of temporaries: as early as possible vs. at statement boundaries
    - in-place updates to symbol table versus immutable persistent symbol table
    - type checker produces output (e.g. types of declared names) as hash tables vs. producing a new AST

2021-06-07

- Attendees: josh11b, mconst, chandlerc, zygoloid
- "& struct literal" constraint syntax
    - separating from the labeled arguments, don't like associating it with function calling since it can't happen multiple times, and is more "membery"
    - can apply either to individual interfaces or the whole expression
    - not associative
    - might not be commutative
    - is the syntax "too cute"?
    - can expose a name hidden when two interfaces have conflicting names
    - Compare with the Swift approach, where associated type conformance is more structural
        - concerns about mixing incompatible interfaces
        - longer names to avoid conflicts
        - programming in the large, particularly evolution without introducing conflicts
    - Concern about evolution from just `Foo` having a `.Type` member and then `Bar` adding one in: `Foo & Bar & {.Type = Baz}`

        - Proposal: & is an n-ary operator, only applies when all the interfaces have the name
        - Consider other operators:
            - (Foo & {.A = Int}) & (Bar & {.B = String})
            - Foo & Bar with .A = Int, .B = String
            - Foo w/ .A = Int & Bar w/ .B = String
            - Foo(A=Int) & Bar(B=String)
            - Foo[A=Int] & Bar[B=String]
            - Foo{.A=Int} & Bar{.B=String}
    - Carbon: external impl and conditional conformance syntax
        - Josh's opinion: prefers type name before interface name
        - Should we define all the unqualified names for a type in its definition, or can it just be in the same file?
        - Example that was interesting from Dave's Swift code, translated to Carbon proposal:

```
struct TupleSyntax(Payload: AST) extends AST {
  // ... common implementation ...
  expand TupleSyntax(Expression) { ... }
  expand TupleSyntax(Pattern) { ... }
  expand TupleSyntax(TypeExpression) { ... }
}
```

        - Chandler: should probably keep all the definitions for unqualified names for a type together even if it is multiple blocks.
        - Chandler: C++ puts it all in one block, would prefer to avoid a familiarity hit
        - Richard: Would restrict to one impl block per type
        - MConst: one place to look for data members, one place to look for methods
        - Richard: Likes that in Rust there is a uniform way of adding methods to types
        - Chandler and Richard: not as interested in supported extension methods
        - MConst: Need to mark explicitly things that are not part of the unqualified names
        - All tentatively liked this syntax:

```
struct Vector(T:$ Type) {
  // data and methods ...

  // Non-conditional conformance, adds to unqualified API
  impl Printable { ... }

  // Conditional conformance, adds to unqualified API
  impl Vector(C:$ Comparable) as Comparable { ... }
  method (me: Vector(C:$ Comparable)*) Sort();
```

```
}

// External impl, only qualified access
external impl Vector(T:$ Type) as Indexable(Int) { ... }

// External conditional conformance, only qualified access
external impl Vector(C:$ Comparable) as Comparable { ... }
```

    - Chandler: Forward declarations can be used to reduce indents
    - MConst: Prefer one way to do things
    - Josh: Maybe just follow what is C++ common practice re: forward declarations, while removing footguns re: incomplete types
- Method syntax

```
struct Vector(T:$ Type) {
  method (me: Vector(C:$ Comparable)*) Sort();

  fn Sort[me: Vector(C:$ Comparable)*; ...]();
  // Vanilla C++ shoehorned to make it have enough info
  fn Sort() Vector(C:$ Comparable)*;
  // This is based on the potential new direction of C++
  fn Sort(me: Vector(C:$ Comparable)*);
  // Go-style method
  fn (Vector(C:$ Comparable)*) Sort();
}
```

2021-06-03

- Attendees: josh11b, zygoloid, geoffromer
- What is the non-type generics/templates desired behavior?
    - Would be nice to have uniformity between type and non-type story
    - If we only had one story for non-types rather than both a generics and templates story, what would it be?

```
// Possibly doesn't need a $ at all,
// even though we might not even be able to evaluate F at runtime
// However we couldn't monomorphise types here with a runtime value for n
fn F(n:$ Int) -> Container {
  if (n == 0) return Vector(Int);
  return Vector(F(n - 1));
}
fn G(n:$ Int) {
  var v: F(n);
```

```
  // ... use Container interface to access v ...
}
```

- We could type check `F` and `G` independently
- These two statements would have very different behavior:

```
var ty:$ Container = F(N);
var v: ty;
// Not okay to initialize a $$ value from the return value of a $ function
var ty:$$ Container = F(N);
var v: ty;
```

- Is it sensible to have a function returning a type called at runtime? Hard to say what a type is?
- Changing the question: is it sensible to use a run-time computed value in a type position?
- Question: do we want to guarantee that we can fully monomorphize if the compiler chooses? Yes, it is always a valid optimization strategy.
- Could possibly still return dynamic types, as long as they aren't used in a type position. For example, a Rust dyn box might support dynamic type values.

```
struct { var c: Container; var p: Void* }
(p as c*)->PushBack(x);
```

- Josh: need this for a couple of Carbon-provided builtin types, but doesn't seem worth providing a general purpose facility for users to do this themselves. Too much work and complexity for the reward.
- In Rust, `c.ElementType` needs to be constrained to a concrete type because `x` can't have dynamic type.
- Concerns about whether you can declare `c`, and whether you can write `as c*` which involves a dynamic type in a type position
- If you had some restrictions, such as only ever use pointers to the type, would it be okay to opt-in to taking a dynamic type? Yes, some operations require a $/$$, but not all.
- Part of the story for writing functions that operate on dynamic types, for space or expressivity reasons.
- If there is a timeline of operations, can use a value from some point forwards? Type not available at all?
- Differently tagged strings to e.g. mark attacker-controlled data, some functions are generic across tags

```
fn F(T: Type, p: T*) {
  // Can't do anything type-specific to p
}
vs.
fn F(T: Container & {.Elt = Int}, p: T*) {
  p->PushBack(3);
  // interpreted as: T.PushBack(p, 3);
```

```
}
// Would this type check?
fn PushBack[T: Container & {.Elt=Int}](T*, Int)
// T is just a table of function pointers
```

- Maybe "no runtime information about type" just means "is of type Type rather than something more specific"
- Can you write `*p` in `F`? What if you had a function like this: `fn Deref(p: T*) -> T` ? If `T` is only known dynamically, can't return a `T` value.
- However `fn Deref[Type:$ T](p: T*) -> T;` is fine
- Would $ be useful for expressing the size of a coroutine frame? It behaves like an object, but its size is not known until the optimizer has run. $ not quite right, distinction between before or after codegen. This weird size is contagious to any struct containing one. Can you put these in arrays? Hard to even come up with an upper bound on the size needed.
- What is a runtime computed type?

```
// F doesn't have any parameters, so they can't be $ or $$.
// Can F be run at compile time?
fn F() -> Type { return Int; }
var i: F() = 2;

var ty1:$ Type = Int;
// i1 fails to compile since it ty1 is only a $ Type and those in general
can't be set to 2.
var i1: ty1 = 2;

var ty2:$$ Type = Int;
// Legal
var i2: ty2 = 2;
```

- Possible rule: for values in type position, we try and evaluate them before type checking, and if that doesn't succeed but is known to be able to be evaluated after type checking, use the type of type, and then evaluate it again after type checking.
- Do we want to require that any generic type expression will evaluate successfully and terminate? Alternative is to accept the possibility of monomorphization failures. Does living with this restriction give up too much expressivity?
- Concern is functions whose signature indicates it returns a type, but are not total. Tough to have an expressive enough sublanguage that is guaranteed total.
- `F()` example is fine as long as it is executed at or type checking time, when we can give failures early enough
- Do we want a declarative statement of intent that a function can be run at compile time? The set of things you can do at compile time is both wider and narrower than those at runtime. Also the body of such functions need to be made available to callers at compile time.

        - From an evolutionary perspective, we don't want people to rely on functions being usable at compile time unless the author is OK with that, because that restricts how the definition can be changed.
        - So, probably do need some marker
    - Concern: if `F` had a $ parameter, it might need to be evaluated symbolically. Interesting example:

```
fn F(t: Type) -> Type compile-time-callable { return t*; }
fn G(t:$ Type) {
  var v: F(t);
}
```

    - Same issue as calling templates with generic arguments. Call to F happens at type checking time, and can fail. Milder form of the same issue without calling a function: `var v: t*`, `t*` is still computed symbolically.
    - "symbolically evaluatable" != "compile-time callable"? Maybe its okay to not be symbolically evaluatable as long as the failure can be diagnosed when executing it during compile time? Concern is function:

```
fn F(t: Type, b: Bool) -> Type {
  if (b) {return t;}
  else {return t*; }
}
```

    - Does the answer depend on whether the value of `b` is known to the caller? We could probably support: `var v: F(t, True);`
    - Much easier to determine if the function is okay when actually trying to execute it with the compile-time arguments during typechecking than determine if it is symbolically safe in general.
    - Termination is not the issue, but defined behavior. For example, forming an array with negative size.
    - What about this?

```
fn F(t: Type) -> Type {
  if (t == Int) {return t;}
  else {return t*; }
}
```

    - Fail if we can't determine if `(t == Int)` is true or false. May still be able to answer the question even if `t` is only known generically, like t = U*.

```
fn Q(t: Type) -> Type {
  // R becomes valid if this line is commented out
  if (t == Int) {}
  return Int;
}
fn R(t:$ Type) {
  var v: Q(t) = 0;
```

```
}
```

```
interface Foo { fn f(); }
struct X { fn f(); { ... } };
impl X for Foo { fn f() { ... } }
fn Q(t: Type) -> Foo {
  if (t == Int) {}
  return X;
}
fn R(t:$ Type) {
  // is Q(t) returning X or X as Foo?
  var v: Q(t);
  // Determines whether this calls X.f or X.(Foo.f)
  v.f();
}

var t:$ Foo = X;
var v: t;
// calls X.(Foo.f)
v.f();
var t:$$ Foo = X;
var v: t;
// calls X.f, or give an error
v.f();

var v: (X as Foo);
```

- Templates are not generics -- is it part of the type or the binding. If it is part of the binding, then what do you do when there is no binding? Disambiguation could be to prefer the template interpretation when it is available.
- The semantics were intended to be able to give an error when the semantics were changing due to switching from templates to generics

```
var t1:$$ Type = X;
var v1: t1;
// calls X.f
v1.f();
// ok?
v1.(X.f)();
// could jump straight to, if you want:
v1.(Foo.f)();

var t2:$$ Foo = X;
var v2: t2;
// an error
```

```
v2.f();
// still ok?
v2.(X.f)();
v2.(Foo.f)();

var t3:$ Foo = X;
var v3: t3;
// calls X.(Foo.f)
v3.f();
// still ok
v3.(Foo.f)();
v3.f();

var v: (X as Foo);
var n: (Int as Type);

// Type checks
fn F1(i: Int as Type) -> Int {
  var j: Int = i;
  return j + 1;
}

// Does not type check
fn F2[T:$ Type](i: T) -> Int {
  var j: Int = i;
  return j + 1;
}

fn F1(i: Int as Type) -> Int {
  var j: Int = i;
  // Not allowed, Int as Type is not Addable
  return i + 1;
  // Allowed:
  return (i as Int) + 1;
}

var v: (X as Foo);
var w: (X as Type);
// Type of X is *not* Type, since (X as Type) doesn't have `f`, but `X` does

var ty:$ Type = Int; // ty is "Int as Type"
var ty:$$ Type = Int; // ty is "Int" (with type-type being Int.Type)

var ty:$ Auto = Int; // ty is "Int" (with type-type being Int.Type)
```

        - It is sort of okay that the type of `ty` is not `Type` because we do the substitution before we do type checking. In some sense the type of `ty` is a bit meaningless since it is only used before type checking, not during.

```
// The interface of ty (and therefore the type-type) is the
// symmetric difference between X and Foo.
var ty:$$ Foo = X; // ty is "X as$$ Foo", with type-type being (X as$$
Foo).Type
```

        - The type-type also has the fields, since you lose the fields when you change the type-type.
    - Methods separate from data
    - Keyword question
        - considering `extends` for 1 and 2
        - considering `:` for 1, but Josh wants a keyword 2
        - going to come up with another solution for 3
    - Inheritance, three main use cases:
        - needed for C++ interop
        - useful to be able to support partial or complete inheritance of implementation of an interface
            - in contrast to the Rust approach, where you have to re-implement the interface and manually forward anything you didn't want to change
        - "base to derived cast": relatively principled way of navigating from a subject to the containing object, undoing some type erasure you've already done; also useful in intrusive linked lists to be able to point to the next next pointer, but then want to go from the next pointer to the object
        - has a practical implementation
        - would allow C++ programmers and program designs to be portable to Carbon
        - C++ vtable pointers inline, less expensive in the case where you have a big data structure holding base pointers
    - Can we build the C++ model of single inheritance with vtables using the interface machinery
        - Reminded of the bit in the initialization doc talking about the type without the vtable pointer
        - Any change to the model will introduce an incompatibility with C++
        - Most inheritance / virtual functions is not ABC, also get a way to define customization points for the base class to call the implementation defined by the derived class
        - Maybe support multiple inheritance, but only the first parent can have data
        - Mixins need to be able to support the intrusive list use case
        - Can we expose interfaces to C++ as ABCs?
            - C++ ABC descendents implement the interface
            - Can automatically create a wrapper object that implements the ABC and holds a pointer to a Carbon type implementing the interface

    - What is the migration story for iostream? ofstream has a virtual base (even though it doesn't need it). Could expose ofstream as a final class with no base classes.
        - interop is harder than migration story: consider a class with a virtual method that takes an ostream reference
        - virtual base class inheritance gives you the API but not the subtyping relationship, preserves the ABI
    - Can we just support virtual inheritance in Carbon?
        - Rule: you cannot directly access the fields of a virtual base class
        - Can call functions, the C++ side will understand how to access the fields
        - May want specific support for iostreams, custom migration tools
    - Certain pointer conversions are not reversible in C++
        - pointer to class to pointer to member of that class is one example
        - reverse is useful and used in practice, but not allowed by spec
        - concern is optimizations that rely on escaping a pointer to one member not affecting cached values for other members
- Information in type or on the side? For example for literals
    - Worried about overflow and forwarding
    - Simplest solution is type of 5 is Int32, and overflow is a compile-time error
    - Don't want implicit widening since it is bad for vectorization; operations happen in the widest type of its operands. Concern is that literals need a narrow type to avoid unintended widening
    - Possible rule: type of 5 is "Int64" but we allow narrowing conversions that can still represent "5". Keeps additional data about values beyond the type.
    - Another approach: don't infer integers types from literals for you
        - Concern: passing an integer literal to a function that takes multiple integer types
    - What is the type of the expression `n + 1`? Same as `n` seems to make sense if we don't want implicit widening. But then what about `n + 500`, if `n` is an `Int8`? (Maybe that could be an error, because, again, no widening.) Even then, edge cases are potentially worrying: `n * 2` might preserve the type of `n`, but does `n * 127` if `n` is an `Int8`? Does `n + 255` if `n` is an `Int8`? Some tricky questions here.
        - We probably want to be able to distinguish between "this is a constant value that had no explicit type information given" and other integer expressions, whether that's a part of the type (eg, `CInt`) or some separately-propagated information that's not part of the type.

2021-06-01

- Attendees: josh11b, chandlerc, wolff, gribozavr, jonmeow
- #524: Generics overview: discussion about reorganizing

- Dependency woes
  - LLVM releases are broken for us
  - Homebrew doesn't install good stuff on Linux
  - People seem to use GCC instead of Clang on Linux
- New constraint syntax for generics:
  - New syntax Generics details 7: type equality for argument passing
  - New syntax Generics 5: constraints
-

2021-05-27 part 2

- Attendees: zygoloid, josh11b, chandlerc, jonmeow
- #478, #505: labels in function calls
  - Leaning toward anonymous struct approach
  - Motivation for the anonymous struct / brace syntax:
    - simplifying the collection of types of values: tuples are positional, structs are named
    - clear and simple story for C++ interop (making the Carbon story worse to match C++)
  - Using `...` in the Python approach to support forwarding

```
fn F(args: (Int, Int, Int)...) {
  let (x: Int, y: Int, z: Int) = args;
   // ...
  G(args...);
}

fn H(args: (Int, Int,  Int)..., named: {a: Int,  b: Int, c: Int}...);

fn K(args: (T:$ Tuple)..., named: (S:$ Struct)...);

fn F(a: Int, rest: auto..., named: {x: Int, y: Int})
fn F(a: Int, rest: auto..., {x: Int, y: Int})
```

  - CAS: Can't specify named parameters without also specifying all optional positional parameters

```
struct X { var b: Int; }
fn H(a: X = 1, {b: Int = 2})
H({.b = 3});

fn J(a: Int, b: Int = 1, {c: Int = 2, d: Int = 3});

fn Q({a: Int = 0}, b: Int = 0);
Q()
```

```
Q({})
Q({},  1)
Q(1) // error
```

    - Since we are using named arguments in place of comments, we are going to have things like:

```
ExportFilesToLogManager(/*removed=*/true);
->
ExportFilesToLogManager({ .removed = true }); ?
```

    - Also the generics use case, which is motivating us to address this question now

```
fn F[T:$ Integral, C:$ Container({.Element = T})](c: C*);
fn F[N:$ Int, C:$ FixedSizeContainer({.Size = N * 3})](c: C*);
fn F[T:$ Integral, C:$ Container({.Element = T})](c: C*, n: T);
fn F[N:$ Int, C:$ FixedSizeContainer({.Size = N * 3})](c: C*, a: Array(N,
Int));

Container{....}
ParameterizedContainer(...){....}

[c: Container]
[c ~ Container{...}]  // <- don't like
fn F[T:$ Integral, C:$ Container(.Element = T)](c: C*);
fn F[T:$ Integral, C:$ Container(Element: T)](c: C*);

c:$ Container & {.Element = T}
c:$ Container & Comparable
c:$ (Container & {.Element = T}) & Comparable
```

- For method syntax, like `me: Self` more than `self: Self` or `this: Self`
    - Gets us closer to eliminating implicit member lookup
    - chandlerc: `fn Method[me: Self; ...](...) {` better if `me` is a keyword
- should we allow both `me: Self` and `me: Self*` for explicit object parameter notation? should we have some kind of `&me: Self*` reference notation?
    - zygoloid wants either `me.x` in both accessors and mutators, or `me->x` in both, for ergonomic reasons
    - chandlerc wants `me.x` in accessors and `me->x` in mutators, so that the differing semantics and especially safety properties are visible
    - zygoloid: we actually use an arbitrary mix of `Sema*` and `Sema&` in clang code, and that's annoying but … has been largely fine in practice
        - but still don't like that refactoring between accessor and mutator requires changing `.` -> `->`
    - zygoloid: don't like the implicit `&` at the call site for mutators
    - chandlerc: similarly, not clear from a safety point of view that the implicit address-of is a good thing

2021-05-27

- Attendees: zygoloid, josh11b, chandlerc, gribozavr
- #478, #505: labels in function calls
  - zygoloid: prefer Python calls, but worried about declaration syntax
- #510: returning without copying
  - using `returned var` may mean we can get rid of NVRO, which would be a net simplification
  - lifetimes with `returned var` more complicated than lexical nesting, but destructive move also messes with lifetimes
  - note that there are some weird cases, see sum type case from "2021-05-25 part 2" as an example; but this is a new set of options available in the `returned var` approach, not really a problem
  - zygoloid found chandlerc's most recent post on the thread compelling
- Question about C++ interop: expose Carbon types to Clang via text or LLVM IR?
  - Leaning: Link a Clang tool into the Carbon compiler. Biggest concern is scaling to multiple foreign function interfaces.
  - Maybe we can do lifetime annotations on the C++ side, then maybe we don't need to support C++ code with no lifetime annotations from Carbon?
- Carbon: safe or just safer?
  - Evolution story: at first unsafe features without annotation, then all unsafe features annotated
  - Can evolve the language to safety more easily than we can mechanically transform unsafe code into safe code, though we will be able to mechanically migrate some
  - Instead of making safe ergonomic & unsafe unergonomic, will make both ergonomic
  - Incentivize human migrating code to safe by performance improvements that require safety.
  - Need a compelling checking story for unsafe constructs, for testing and hardening, more so than Rust needs.
  - Incremental, progressive narrowing of which code is unsafe. Allows decoupling the language transition from the safety transition.
  - May want to target bug classes independently of the region code. Example: lifetime safety before addressing data races.
  - Still need syntactic warning signs to clearly delineate unsafe.
  - Mechanically add unsafe markers once safe patterns have good adoption.
  - Rust uses blanket unsafety, but maybe Carbon would label what specific unsafety
  - Like it when the code validating that unsafe code meets safety conditions is near the unsafe block.
  - Incrementality of adding safety attributes almost forces granular safety annotation.

- #523 Pointer syntax (and #520 whitespace-sensitive operator fixity)
  - Closer to a resolution on the syntax than the disambiguation approach.
  - `Ptr()` is painful
  - Options for using * for pointers, deref, and multiplication
    - marker, delimiter to switch to type expression, only get postfix-* for types, no where else
    - whitespace disambiguates
    - disambiguation rule in a conventional grammar; sometimes incorrect but can be overridden by `()`
    - no ambiguity in grammar
  - Would like to keep grammar LALR(1) or LR(1) + rules in the lexer
  - Bounded lookup if you treat a sequence of stars as a single token
    - no spaces allowed between consecutive stars?
  - Likely will need to either forbid or require whitespace
  - Issue is frequency of conflicting with what people want to write, important with a*b
    - chandlerc interprets x^5 as exponentiation vs. x ^ 5 looks like bit-xor
  - Worried about teaching it. Bad: "It does what you mean except there is this long doc with rare edge cases."
  - For Swift, syntax highlighting doesn't require the compiler because you don't need to figure out what the operators are, just that these characters get the operator characters. Have to use the compiler for anything more complicated like refactoring.
  - Hard tradeoff here
  - What could we do with just 1 symbol of lookup?
    - Allow `a*b` but not `a**b`,
  - Could look on both sides of the `*` in the lexer and have a table of cases
    - cases to consider: `)*+ )*- )*=`
    - What is the default in the non-obvious cases?
    - identifiers, literals, parens
    - asymmetry between [and]: x[i]*3 vs. x[i]*[3]
  - The rule:
    - LHS:)}] identifier literal
    - RHS: (identifier literal
    - If the thing on the left is in LHS and the thing on the right is in RHS or whitespace on both sides, then binary operator, else unary
    - Still disallow whitespace between unary operator and operand
    - Don't allow space in `x[` or `f(`
  - Doesn't allow angle brackets, such as `X<3>()`
  - Maybe treat `*` and `/` (maybe bitwise `&`) differently from other operators; when there is a precedence-based reason; to make the exception narrow
  - Int-space-*-space -> multiplication, Int-space-*-x -> deref, probably an error
  - `var x: Int * = 0;` -> error, needs to be rewritten to `var x: Int* = 0;`

2021-05-25 part 2

- Attendees: chandlerc, zygoloid, jorgbrown, josh11b
- Discussion re: #510: Should we have named return variables, returned declarations, or both?
    - Reason for and against named return variables is similarity to C++ constructors
    - Consider alternate syntax:

```
fn MyClass.Make() -> Self {
  // need to alert that lifetime is different
  var self: return = {.a = 1, .b = 2, .c = 3};
  if (...) {
  }
}
var return self = {...}; <-- need to alert that lifetime is different
return var self = {....}; <-- bad unless control flow
returned var self = {...};  <-- and "returned let"?
retvar self = { ... }; <-- and "retlet"?

// if we support Swift-style if var ...:
if retvar self = ... {
  ....
  return;
}
```

- Aside: `let` vs. `var` vs. `var $`
    - C++ `const` vs. `constexpr` vs. `consteval` vs. `constinit` vs. const integral expressions
- `if var`:

```
alias Result(T:$ Type, E:$ Type) = .Success(T) | .Error(E);
fn F() -> Result(Int, String) { ... }

if var .Success(i: Int) = F() {
  A;
} else {
  B;
}
// equivalent to:
match (F()) {
  case .Success(i: Int) => { A; }
  default => { B; }
}
```

- Combine with `match`/`case`? An argument to instead using a keyword (maybe even `return`) for the type, but...
- Problem with `var i: return` is that it doesn't compose like types do:

```
var i: return = F();  // Okay
var (s: String, i: return) = G();
var t: (String, return) = G();
// Problem, can't write: &t since the String and the return are in different
places.
```

- The thing that indicates that it is special has to be with the introducer
- Should be able to say:

```
fn H() -> (String, Int) {
  returned var (s: String, i: Int) = ...;
}

fn J() -> Result(Int, String) {
  if (...) {
    returned var .Err(s: String) = .Err(GetErrorMessage());
    return;
  } else {
    returned var .Success(i: Int);
    i = 3;
    return;
  }
}
```

2021-05-25

- Attendees: josh11b, chandlerc, jonmeow
- Templates in generics design proposals?
  - Few options to choose from, probably just include template content in a follow-up so we can keep these proposals smaller and more focussed
  - Chandler suggests that we probably do want Carbon templates in addition to C++ template interop, but will keep it as a separate question.
- Frustration installing LLVM on Linux via HomeBrew
- Trying to collect data about the impact of memory safety bugs
- Also interested in collecting data about the amount of C++ code is performance-sensitive at Google

2021-05-24

- Attendees: chandlerc, zygoloid, mconst, josh11b
- Pointer syntax

    - C++ inside out: bad; not even in a single location
    - zygoloid: If we want to have type functions like `Vector(Int)`, and a consistent type reading order, it would have to be left-to-right
    - mconst: Still might be okay to have inconsistent reading direction (like expressions in general) as long as it is reasonably intuitive
    - C/C++ types start with a word that is suggestive of types
    - Using `()` instead of `<>` in function calls doesn't clearly separate types from other values
    - References? Do we need 2 pointer syntaxes?
- What do we need from a safety perspective?
    - How are we going to transition C++ code?
    - Could we transition C++ code if we were not worried about enforcing exclusivity, just lifetimes & fixing use after free?
    - Worried about the case of a struct with a lot of interior pointers and hard to express & possibly different lifetimes.
    - Two big problems, but currently lifetimes are a bigger security problem than races.
    - What do we need references for? Operator `[]`, match statements, mutable this, maybe efficient argument passing
    - C++ reference -> Carbon pointer ; C++ pointer -> Carbon optional pointer
    - Two different operators []: Get element, put element ? Plus GetPut and address of?
    - chandlerc: would prefer that the interface to access to an element of an hash map was not in terms of passing pointers to elements around
    - mconst: can we do this in terms of simple rewrites to a small set of functions?
    - different hash map implementations: one that never exposes pointers to elements, and another node-based one that never invalidates pointers
    - iterator on a sorted container maybe is only available after moving from the mutable container API and restricting to an immutable API
    - would need Rust take for things in a data structure, Swift inout only moves it notionally
    - goal is to make time to do safety research to evaluate a lot of alternatives
    - if we have unsafe containers and safe containers, perhaps the unsafe containers are the C++ containers? Concern that we can still improve on C++ containers a lot before we get to safe containers
- Back to pointer syntax:
    - Concern is multidimensional arrays, most arrays are single dimensional
    - Arrays are probably less common than vectors and slices
    - Maybe just `Array(N, T)`
    - Chandler: postfix-`*` makes pointer type, prefix-`*` dereference, whitespace disambiguation
    - default is infix, whitespace or a collection of symbols (`:)],`) after a symbol can make it postfix
    - question: is `x*y` legal?

    - zygoloid: whitespace on both sides makes it infix, otherwise it is unary. Interesting examples:
        - `Vector(Int *)`, `fn(p: Int *, q: Int *)`
        - chandlerc: would choose a more restrictive rule
        - zygoloid: concern that the `*` might be hard to read
    - zygoloid: can figure out the whitespace in the lexer, no problem in the parser
    - Whitespace rule:
        - binary operators must have space on both sides
        - prefix operators must not have a space after
        - postfix operators must not have a space before
        - unary operators might not any whitespace on either side
        - may need some experience to get the right rule for "did you mean" diagnostic
        - expect to be able to recover in common cases
        - need to forbid expression followed by expression
        - need to forbid function call with a space `F (x)` including:
          ReallyLongFunctionName
            (arg1, arg2, arg3, arg4, arg5);
          Need to put the `(` on the previous line (unless it is a function declaration)
        - This rule allows us to use `<..>` as angle brackets
        - Also gives us `<-` and `<=` operators; these probably would forbid `a+b` binary ops without spaces around them
        - `*(*x)` vs. `**x`: probably have to allow the latter, no max-munch rule
    -

2021-05-20 part 2

- Attendees: zygoloid, josh11b, chandlerc, mconst
- #478 and #505: labeled parameters/arguments
    - Some concerns about the set of possible completions after a `,` in an argument list
    - CAS curly anonymous struct approach:
        - F(a: Int, b: Point = {.x = 2, .y = 3}, {.x: Int = 4, .y: Int = 5});
          // ambiguous:
          F(6, {.x = 7, .y = 8});
        - Could model this as a last optional positional anonymous struct parameter with an empty default
        - More flexibility to pass in a struct value in for this last parameter
    - "Python" approach:
        - F(a: Int, b: Point = {.x = 2, .y = 3}, named x: Int = 4, .named y: Int = 5);
          F(6, {.x = 7, .y = 8}): // sets `b`
          F(6, x = 7, y = 8); // `b` gets default value

        - Still choices for the declaration syntax
        - Some preference marking each parameter rather than saying "from this point forward all parameters are named"
        - fn F(a: Int, b: Point = {.x = 2, .y = 3}, x=: Int = 4, y=: Int = 5);
    - Optional requires named approach:
        - Use `= _` to mean named & required
        - Assumption is named & optional is the common case
        - Uses an `=` in the declaration when you use an `=` in the call
        - If it is optional in C++, it becomes named in Carbon
        - Can use overloading as needed if necessary in rare cases to represent positional parameters with defaults, or just in the C++ interop layer
        - Would use the `...` operator to splat a struct into the argument list
        - Could use `F(6, arg::x = 7, arg::y = 8)` in C++; good for boolean parameters, and makes your natural Carbon API design work in C++
        - Could very well want positional optional parameters for C++ interop to still be accessed via names in Carbon code
    - Questions to ask:
        - What should the named call syntax look like in Carbon? Python or anonymous struct
        - If the Python approach, how should we declare that in Carbon? Jon's approach with =s, `named` keyword, or fill in the blank suggestion
        - If the Python approach, how should we call these functions from C++? an attribute that turns named parameters into positional parameters (possibly with defaults), `arg::` approach, option struct
        - How do we migrate C++ code that uses optional parameters to Carbon, such that it can be called the same way from C++? could be overloads, or the attribute from the previous question that allows them to be used with names from Carbon.
- Carbon: pointer syntax choices
    - Dereference is going to be more common than declaring pointer types, and so is more important.
    - Prefix-`*` for dereference is still on the table, but there was support for a postfix dereference operator using an otherwise unused symbol (`!@#$^` or others).
    - If we choose `^` for pointers, we need another story for bitwise `^`. Two options: binary/infix `~` could be bitwise xor, or we could switch the bitwise operators to two characters (`\&`, `&:`, or something).
    - jonmeow: Concern that `^` is harder to type than `*`.
    - Desire to use a symbol for pointer types assuming that they are going to be commonly used when calling functions. If we use a postfix-symbol for dereference, using it as a prefix would be reasonable to make a pointer type if we don't have another use for that symbol (there was a concern that prefix-`@` might be used for attributes/annotations).

    - zygoloid: Prefer `Array(N, T)` over `Array(T, N)` for the same reason that postfix `[#]` is confusing when used to declare multi-dimensional arrays.
    - zygoloid: Even if we have prefix-* dereference, perhaps we should also have a postfix operation

2021-05-20

- Attendees: zygoloid, josh11b
- Looking like some possible consensus on #542
- Review of survey data
- How do you feel about dots in named parameters?
    - zygoloid: Problem with requiring `;` in call sites to match `;` in function declarations, and that indicates a separation between ordered and unordered arguments independent of whether they are labeled. Suppose we start with:

```
fn F(Int: a, Int: .b = 1);
F(a, .b = 3);
```

You might not bother using a `;` instead of a `,` here to indicate that `.b` is reorderable, because there's only one named optional argument. Then `F` evolves to have a second one:

```
fn F(Int: a, Int: .b = 1, Int: .c = 2);
```

`F` must make an unfortunate choice: either they use `;` now, and break all call sites, or don't, and don't allow reordering.

    - zygoloid: Suggested solution: when you write a function call, you are always giving the arguments in a particular order, separated by commas uniformly, and it is the declaration's pattern that says how it unpacks that into ordered and unordered parameters.
    - josh11b: I'm not very interested in having both positional and unordered labeled parameters, and that makes the `;` just redundant encoding of information already passed.
    - josh11b: Would like to make restrictive choices now and see how many we can live with.
- josh11b expressed rationale for N:TD > TND > T:ND
- Carbon: pointer syntax choices
- We can erase differences between built-in types and user-defined types by allowing users to implement interfaces for built-in types, but the type declaration syntax for built-in types may put it on unequal footing. Should be done only when it is the right choice.
    - pointer vs. `unique_ptr`/`box`
    - span/slice > vector > fixed-sized arrays > small vectors
    - hash_maps have a lot of implementation strategies, unlike vector; Python, Go, Perl, etc. chose to build one in, but they have less focus on

performance; could possibly pick one that is "good enough". Different APIs actually admit different implementation strategies. Is there a hash_map implementation that is good enough that you should almost always use it?

2021-05-17

- Attendees: chandlerc, josh11b, jonmeow, mconst, zygoloid
- Carbon: designators as symbols
  - Lots of details to work out if we want to go this direction
  - Doc does give some reasonable semantics to what designators could mean more generally
  - Symbols have been inherited through languages, e.g.: Lisp -> Smalltalk -> Dylan -> Objective C
  - In Swift, protocols are collections of symbols, and extension methods
  - These ideas seem potentially useful for compile-time metaprogramming.

```
var auto x = .None;
// x has type .None, a zero-sized type.
var Optional(Int) y = x;
// Not allowed: x = y;
```

- Carbon: labeled params brainstorming
  - Chandler: Possibly could say: no destructuring of named tuples in variable declarations; Richard: not happy with the options here
  - When returning, more interesting to destructure positionally
  - Instead of "named tuples" maybe "unnamed structs"?
  - Richard: non-positional/named -> more discardable, particularly when looking at return values
  - Three ways to indicate labeled:
    - separator between positional and named
    - marking names
    - having a default value, which may be `required` if there is no default
  - Richard: leaning toward positional parameters by default unless optional or ambiguous (like bool params), due to C++ heritage and non-sentence
  - Not much motivation for deep matching in function parameters or variable destructuring.
- Lots of discussion about pattern matching
  - Looked at examples from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf
  - Asymmetry of expressions and patterns, for example the difference between allocating in an expression and dereferencing in a pattern
  - Given a 3-tuple there are lots of ways of interpreting it, but given a color, there may only be one way to interpret it as a 3-tuple.

- Whitespace sensitivity around operators
  - No rules for configuring whitespace around binary operator in Clang; Clang-Format Style Options — Clang 13 documentation

2021-05-13

- Attendees: chandlerc, josh11b, jonmeow
- chandlerc: Liked extract((.key, .value), F()) idea just to give space / options
- chandlerc: Lots of options, can be subtle, especially if we employ multiple
  - local marking (e.g. adding a keyword or a `.`)
  - separators (`named` or `;` instead of `,`)
  - defaults only on named arguments (or a `required` if named but no default)
- In person meetup? Back yard or park?
  - Cuesta Park in Mountain View has shaded picnic tables
  - Baylands Park is close to the Sunnyvale office and is more centrally located.
  - Likely many other options, like Lakewood Park
  - Back yard might be better if we want to work on laptops, though
- Lambda syntax using `$`?
  - `$(x + y)`
  - `$($1 + 42)`
  - `$[...](....)`
  - Discussion about Ruby : block as parameter and Swift: Trailing closure , note limitations of Swift trailing closures and that they added support for multiple trailing closures (motivation) and labels for trailing closures
  - Zig `catch` operator:
    - `a catch b`
    - `a catch |err| b`
    - If `a` is type `Result(T)`, then `b` must be of type `T` or `"noreturn"`
    - Type of the result is `T`

2021-05-10

- Attendees: jonmeow, josh11b, mconst, chandlerc, zygoloid
- Some Git scripting, and talking about Git workspaces
- #505 Swift (declarer decides) model vs. Hybrid model?
  - Agreement that we should support positional-only parameters, significant names should be opt-in.
  - Agreement that we have good use cases for required-labelled parameters and they should not be ordered
  - josh11b, mconst, zygoloid argued for Swift over Hybrid; chandlerc doesn't have strong feelings but is willing to defer to the others.
  - What about use cases 5 & 6? Just like we are not worried about 2, not worried about 5. Argued that named elements are the common case.

    - chandlerc would consider no named parameters, and instead using a named literal option struct like Go or JavaScript, but is willing to optimize that experience if we believe it is common or important
- #478 : chandlerc prefers A > B > C
    - `(.key = "widget", .value = ....)`
    - `var (.key = String key, .value = auto: value) = Table.lookup("foo");`
    - `var (.key = key : String, .value = value : auto) = Table.lookup("foo");`
    - `var (.key: String, .value: auto) = Table.lookup("foo");`
        - As syntactic sugar for previous line.
        - `name: Type` is the canonical way to write a positional parameter
        - `.name: Type` is the canonical way to write a named parameter
        - With defaults:
          `var (.key = key : String = "ABC", .value = value : auto = 3) = Table.lookup("foo");`
          `var (.key: String = "ABC", .value: auto = 3) = Table.lookup("foo");`

```
struct S {
  var Int x = 1;
  var Int y = 2;
}
var S s = (.x = 3, .y = 4);
fn F(named Int x, Int y);
F(.x = 3, .y = 4);
var (named Int x, Int y) = ReturnsAPair();
```

- `fn Distance(Point(.x = Int x1, .y = Int y1), Point(x. = Int x2, .y = Int y2)) -> Double;`
- Do we support refutable patterns in function declarations? Do we do static or dynamic dispatching? Do we want to support overloading on anything other arity?
- Do we need overloading just primarily for interop with C++?
- This would be a lot simpler except for providing default values. Defaults have other problems like what scope are they evaluated in? Would in some cases be nice to use things like data members of `this`.
    - One possible simplification is that things with defaults can only be `Optional` types, and the default is always `None`. Passed in arguments would automatically be wrapped. Maybe use a `?` to indicate value is optional, except that in normal pattern substitution logic `?` should mean "must not be none". Alternatively, there would be an "optional" keyword.
    - One use case to consider for defaults: context object with file/line/etc. of caller. Sometimes want to forward a context object (when forwarding from an API function to an implementation function), sometimes want to override with an explicitly constructed value (e.g. in generated code).

- Hardest use case is #6. What if we wanted this to be the code we wrote for that case?

```
var (Int x = .a, Int y = .b) = F();
// equivalent to:
var auto TEMP = F();
var Int x = TEMP.a;
var Int y = TEMP.b;
```

```
match (F()) {
  case (.a == 4, .b == 5) => ...
  case (Int x = .a, Int y = .b) => ...
}
```

- Problem: `fn F(Int, Int)` doesn't mean `F` takes (3, 4), it only takes literally `(Int, Int)`.
  Exotic situation where I want that behavior:
  fn Cast[Type$ U, CanCastTo(U)$ T](T x, U) -> U { return x as U; }
  Assert(Cast(3, Float) == 3.0);
- `var Int = F();` "Error: initializer `F()` of type `Int` does not match pattern `Int`."

Have 4 cases

- Variable declarations, which may involve destructuring
  - No destructuring
    - `var Int x = F();`
  - Destructuring an unnamed tuple
    - `extract F() => Int x, Int y;`
    - If we allow no types `extract F() => x, y;`
    - Prefer `extract` to `assign`, `destructure`
  - Destructuring a named tuple/struct:
    - `extract F() => Int x = .a, Int y = .b;`
    - Not preferred option: `extract F() => Int x = _.a, Int y = _.b;`
    - No types? `extract F() => x = .a, y = .b;`
    - Javascript-style syntactic sugar for when the names match:
      `extract F() => x, y;`
      with types: `extract F() => Int x, Int y;`
    - Actual Javascript: `let {x, y} = F();`
      - Short for: `let {x: x, y: y} = F();`
  - Think we can probably do better than this `extract`.
- Refutable variable declarations, like `if let`, etc. which involve unwrapping/refutable destructuring
  - `guard let Ok(Int x) = F() else Err(y) { return Err(y + "additional context"); }`
    `guard let Ok(Int x) = F() else /* we don't return, but come up with some other value for x */;`
    `let Int x = F().UnwrapOrElse(|y| { compute alternate x from`

`y });`
Zig's `a catch b` and `a catch |err| b` syntax is nicer in josh11b's opinion.
    - Should follow the `match` syntax below.
- Function declarations, which may involve defaults
    - Want to support: ordered unlabelled positional parameters, tail may have defaults; then labelled unordered parameters, any subset with defaults
    - No catastrophic failure if you forget the parameter name
    - Ignoring for now: compile-time vs. dynamic, implied vs. explicit
    - `fn F(x: Int, y: Int = 0, named w: Int, named z: Int = 0)`
    - `fn F(x: Int, optional y: Int, named w: Int, named optional z: Int)`
    - `fn F(Int x, Int y = 0, named Int w, named Int z = 0)`
    - `fn F(Int x, optional Int y, named Int w, named optional Int z)`
    - `fn F(x: Int, y: Int = 0, .w: Int, .z: Int = 0)`
    - `fn F(x: Int, optional y: Int, .w: Int, optional .z: Int)`
    - Ambiguity: is `w` a member of `Int`?
        - ~~`fn F(Int x, Int y = 0, Int .w, Int .z = 0)`~~
        - ~~`fn F(Int x, optional Int y, Int .w, optional Int .z)`~~
    - `fn F(x: Int, y: Int = 0, named, z: Int, w: Int = 0)`
    - `fn F(x: Int, optional y: Int, named, z: Int, optional w: Int)`
    - `fn F(Int x, Int y = 0, named, Int z, Int w = 0)`
    - `fn F(Int x, optional Int y, named, Int z, optional Int w)`
- Match statements, which does runtime dispatch
    - josh11b wants prefixes before values like `== 3`, `> 3`, `in (2, 3, 5)`, `in 1..4`, etc. for "value match"

```
match (y) {
  case > 17 => return y;
  case < 2 => return y + 1;
  case in 3..6 => return 7;
  case in (7, 11, 13) => return 8;
  case 8 => return 9;  // Is this ambiguous or do we have to write == 8?
}
```

    - mconst controversial statement: Can only unwrap one layer, write nested match statements instead when needed. So for example, you wouldn't be able to match inside a pair:
      ~~match (ReturnsAPair()) {~~
      ~~case (x > 3, == 4) => return x;~~
      ~~}~~
      ~~match (ReturnsAPair()) {~~
      ~~case (==1, ==1) => return 2;~~
      ~~case (_, ==1) => return 1;~~
      ~~case (==1, _) => return 1;~~
      ~~case (_, _) => return 0;~~

~~}~~
~~match (ReturnsAPair()) {~~
~~ case (==x, ==x) => return 2;~~
~~ case (_, ==x) => return 1;~~
~~ case (==x, _) => return 1;~~
~~ case (_, _) => return 0;~~
~~}~~

- Dynamic cast case

```
struct Apple extends Fruit { ... }
struct Banana extends Fruit { ... }
var *Fruit fruit = ReturnsAFruit();
match (fruit) {
  case var *Apple a => return a->Color();
  case var *Banana _ => return "yellow";
}
```

- Sum type case

```
match (ReturnsAResult()) {
  case alt Ok(x: Int) if x > 3 => ...;
  case alt Ok(Int x) => ...;
  case alt Err(y) => ...;
}
```

- Maybe instead distinguish enum constructors, with different introducers.
- Could drop `case`. Or could only use it for the value match.
- MConst prefers using `case` for both the first and enum case, `case == 8` and `case Ok(Int x)`

- #523 Postfix * for making pointer types only superficially similar to C++, breaks down as soon as you start using e.g. arrays: `Int[]* Int[3]*` or `Int*[] Int*[3]`.
  Prefix * is pretty common: used by Rust, Go, Zig, Jai
  Means types read left-to-right
  - What languages don't use C/C++ inside-out variable declarations, where modifiers are attached to the declarator, and use a symbol to mark types as pointers?
  - BBC_BASIC: Leading ^
  - C#: Trailing * for unsafe pointers
  - D: Trailing *
  - Delphi and Pascal: Leading ^
  - Go: Leading *
  - Rust: Leading * for unsafe pointers
  - Zig: Leading *

How would you write a function that accepts a function type (with varying signatures) as a parameter?

```
// One input
fn F1[Type$ U, Type$ V](fnty(U)->V, fnty(U)->V f);
// N inputs
fn F[Int$$ N, NTuple(N, Type)$ U, Type$ V](fnty(U...)->V, fnty(U...)->V f);

fn G(Int x) -> String;
F(fnty(Int)->String, G);
fn H(Bool x) -> Float;
F(fnty(Bool)->Float, H);

// Using an interface
fn Call(FnInterface$ T, T f, T.Arg a) -> T.Result {
  return f(a);
}
```

2021-05-04

- Attendees: chandlerc, josh11b, jonmeow, jsiek
- Generics details 3: type-types
  - Skip template stuff, say no structural method matching for generics
- Generics details 4: extending/refining interfaces
- Generics details 5: associated types and interface parameters
- Generics details 6: coherence and name lookup
  - Explain problems with no coherence
  - Replace single letter names with specific examples
-

2021-05-03

- Attendees: jonmeow, chandlerc, josh11b, zygoloid
- Difficulty of parsing `var` declarations and parameter lists without a `:`
  - AST model presumes a relationship between AST nodes and tokens
  - Maybe add "synthetic nodes" into the AST model of the current Carbon parser

12 nodes:
`fn F ( Int a , Int b , Int c )`

Tree structure:

- FunctionDeclaration `fn`
  - DeclaredName `F`
  - ParameterList `(`
    - ParameterDeclaration `a`
      - NameReference `Int`
    - ParameterListComma `,`

        - ParameterDeclaration `b`
            - NameReference `Int`
        - ParameterListComma `,`
        - ParameterDeclaration `c`
            - NameReference `Int`
        - ParameterListEnd `)`

Array order (node type, token, node size):
DeclaredName `F` 1
NameReference `Int` 1
ParameterDeclaration `a` 2
ParameterListComma `,` 1
NameReference `Int` 1
ParameterDeclaration `b` 2
ParameterListComma `,` 1
NameReference `Int` 1
ParameterDeclaration `c` 2
ParameterListEnd `)` 1
ParameterList `(` 10
FunctionDeclaration `fn` 12

- Formatting of generics slide, is it readable?
- Discussion about argument passing style of specifying constraints for generics

Syntax oddities of parameterizing interfaces to constrain associated types...

```
interface Point {
  var Int$ N;
}

alias Point1 = Point;

// maybe used as: Point(N: 2)

struct Point {
  var Int X;
  var Int Y;
}

var auto p = Point(X: 42, Y: 13);
// (X: 42, Y: 13) as Point;
var Point p2 = (X: 42, Y: 13);

struct GenericPoint(Number$ T) {
  var T X;
  var T Y;
```

```
}

var GenericPoint(Int) p = ?????(....)


interface Container {
  var Type$ Elt;
  var Iterable(Elt)$ Iter;
  var Container(...)$ Slice;
  ...
}

alias StringContainer = Container(Elt: String);

StringContainer(Iter: ...)

alias StringContainer = Container where .Elt == String;
StringContainer where .Iter = VectorIter(String);
Container where .Elt == String where .Iter = VectorIter(String);

Container where (.Elt == String, .Iter = VectorIter(String));

alias StringContainer = (.Elt: String, .Iter: VectorIter(String)) as
Container;
```

- References?
- Will Rust ever convert a shared reference to a type with no interior mutability into pass by copy? https://play.rust-lang.org/?version=stable&mode=release&edition=2018&gist=6a5cf51bcfb141b16a331b5dbf62546d
- Borrow checker?
  - Three pieces: Asserting that a referenced value is alive ("lifetime bounds", "lifetime enforcement"), ownership, exclusivity
  - Single-threaded lifetime enforcement is most valuable piece
  - Exclusivity most expensive ergonomically, affects the shape of APIs vs. C++
- Chandler suggests we might want:
  - Compiler is allowed to copy or not (passing a pointer instead)
  - Callee is not allowed to rely on being able to observe changes
  - Prevent using this with types with `mutable` members
  - We would embrace the fact that it is unsafe, but that const references in practice don't observe changes
  - Sanitizer could detect problems
  - Parameters and local constants are different from struct members and local variables. First case: compiler freedom; second case: user specified. Notice that a local constant ("let") is different than one in a struct -- the value of the latter isn't allowed to be dependent on anything.

2021-04-29

- Attendees: josh11b, jonmeow
- Call syntax with named arguments, https://en.wikipedia.org/wiki/Named_parameter
    - Python: Foo(size = 3)
      Swift: Foo(size: 3)
      fn Foo(Int size = 3) { ... }
      fn Foo(size: Int = 3) { ... }
      var Int size = 3;
      var size: Int = 3;
      Swift: func Foo(length size: Int = 3) { ... }
    - Josh doesn't want: fn Func(size: Int size = 3)
        - Jon asks: "size:"?
        - Label required, no default: fn Foo(size: Int size);
        - Label forbidden, default: fn Foo(Int size = 3);
        - Swift-style:
            - Label required: fn Foo(size: Int);
            - Label forbidden: fn Foo(_ size: Int)
    - Underscore issue
        - fn Foo(Int _)
        - fn Foo(Int _ actual_name)
            - Foo(int /*actual_name*/)
        - fn Foo(_ Int actual_name)
    - Jon proposal:
        - fn Foo(Int length, Int width = 2) means:
            - length is a required parameter, *cannot* be labeled
            - width is *optional*, and if assigned, *must* be labeled
        - fn Foo(Int length, Int width) overload requires calls without labels
            - Allows Foo(0, 2) and Foo(0, width=2) to both be valid calls, *when overloading* the above Foo(Int length, Int width = 2) signature.
        - fn Foo(Int length = uninit, Int width = 2) means:
            - length is required, *must* be labeled
            - width is *optional*, *must* be labeled
            - "uninit" could be replaced by "required" or similar
        - A given function signature may not be called with *optional* labels: it is always *required*, or *disallowed*
            - Believe this is a split from Swift (am wrong, Swift docs are just ambiguous)

```
var (c: Int e, d: Int f) = F(a: 3, b: 4);
// following the fact that you would write a named tuple like
// (c: 1, d: 2)
```

```
var auto x = F(a: 3, b: 4);
var Int e = x.c;
var Int f = x.d;
```

        - Concern: destructuring
          "`F` returns a tuple `(c: Int, d: Int)`, we bind new name `e` to the first component, and `f` to the second. Equivalent to:
          var (Int c, Int d) = (c: 1, d: 2)
          var (c: Int, d: Int) x = (c: 1, d: 2);
        - var (c: Int e, d: Int f) =
          - var (c: Int e, d: Int f) x =
          - var (c: e: Int, d: f: Int)

```
var (Int e, Int f) = (x.c, x.d);
```

```
var (Int e, Int f) = F(a: 3, b: 4)[.c, .d]
```

    - Swift (no names): https://www.hackingwithswift.com/example-code/language/what-is-destructuring

2021-04-27

Generics deep dive

- Generics 4: more advanced usage
  - Binding of $ was a surprise
- Generics 5: interfaces
- Generics 6: facet types
  - [zygoloid] It would be helpful to have a crisp distinction between facet types and adaptor types.
    - conversion between adaptor types is somewhat more explicit
    - the adaptor type equivalence class has nested equivalence classes for facet types of each adaptor type
    - the facet type equivalence class supports arbitrary casting at all nesting levels(?), the adaptor type equivalence class has the hash map problem
      - `HashSet(FooByName)` might not satisfy the invariants of `HashSet(Foo)`
      - but `HashSet(Foo)` really means `HashSet(Foo as Hashable)` after the argument is converted to the parameter type, so `HashSet(Foo as Bar)` is not merely convertible to `HashSet(Foo)`, it's actually the same type [right?]
  - [geoffromer] Slide title "Facet types have the API of the interface" should be "..of the type type"

    - [zygoloid] Would like different terminology for "check that type foo properly implements the interface bar (eg, defines all the right associated things)" versus "check extrinsically that type foo implements the interface bar (eg, look up an impl and make sure one exists but don't look inside)" for pedagogical purposes.
    - [zygoloid] Could we get rid of $$, and instead have a `Template` type that represents a sort of union of all possible interfaces, so `Type$$` becomes `Template$$`, `MyInterface$$` becomes `(Template + MyInterface)$$`?
    - [geoffromer] I don't think that works, because `Template` would have to contain all possible operations from the callee's side, but no operations from the caller's side. But what if it was a `template` keyword that takes the syntactic position of a type type, much like `auto` takes the syntactic position of a type. Also like `auto`, it would stand for a concrete type type that's deduced from the context, and specifically from the function body, because the function body implies a set of constraints on the parameter.

2021-04-26

- Attendees: josh11b, geoffromer, zygoloid, jonmeow
- Function vs method syntax
    - [geoffromer] If only difference is call site syntax, prefer that function syntax generalizes notion of pattern matching the call. If methods are more different, then syntax should be more different.
    - [zygoloid] Desirable characteristics:
        - Easy to scan for function name
            - [geoffromer] All else being equal. Tension between this and correspondence between declaration and call syntax.
        - Correspondence between declaration syntax and call syntax
        - Can specify type for 'this'
            - [josh11b] Need to distinguish between pass by pointer and pass by value, going beyond that (eg, C++ explicit this) may be less desirable
            - [zygoloid] Can we get away with only distinguishing those two?
            - [josh11b] Not safe to assume that, but there's a cost in allowing more.
        - Syntactic difference between pass 'this' directly and indirectly
            - [zygoloid] Would like to avoid sometimes implicitly taking the address based on whether this type is a pointer
            - [josh11b] Only want single encoding for "`this` is a pointer" and "implicitly take the address on a call"
            - One possibility: add references back
        - Reasonably readable for the uninitiated
    - Concern with AEHJL: `.` vs. `->` does not reflect whether the caller used `.` or `->`. Could use two other symbols instead? `&`, `=`, `*`?

        - Out of line syntax would be `fn &Foo.F() ...` or `fn Foo.&F() ...`
        - Could in principle use `fn Foo.->F()` but `fn Foo..F()` seems very bad
    - ACFIL `fn (Self* this)->F()` and `fn (Self this).F()`
        - Don't want `fn Foo.(Self this).F()` when defining out of line
        - `fn (Foo this).F()` might work
    - geoffromer: I like something like `fn .F(Self s, Arg arg)` and `fn &F(Self* s, Arg arg)`
        - Pluses: easy to scan for name, clear understanding of how to to treat a method as a function
        - Concern is distinguishing a factory function taking `Self`, like `fn Copy(Self self)->Self`.
    - `a.b(args)` looks up `b` in the type of `a` to find a function `f`, then rewrites as `f(a, args)`
      fn F(Foo, Bar) { ... }
      struct Foo {
       alias b = F;
      }
      `a.b(args)` -> `(typeof(a).b)(a, args)`
    - Parameter passing: do we need to support r-value references? Probably yes, probably will want to support a third option for this
    - Long discussion about references
        - Suppose we have no references whatsoever, do we still want a reference binding mode, eg for parameters? Are references in the type system, or just a property of parameter bindings?
        - `fn F(*(Int* p))`
        - What about returning a reference from operator `[]`?
        - Could have an auto address-of on init reference with explicit deref used for `this`, and other kind of ref explicitly inits to a pointer, but unconditionally implicitly derefs for returning from `[]`.
    - Another option: mutating methods take a pointer to the left of the `.`, do not automatically take address of. Would make which methods could mutate clearer at the call-site.
    - `write Setter(Int n)`, `read Accessor() -> Int;` better than `Me* Setter(Int n)`, `Me Accessor() -> Int;` where `Me` is used elsewhere as the name for the type of `*this` instead of `Self`.
    - Conclusions:
        - We don't want the fully elaborated type to the left of the name of the method. We either want different introducers for mutating vs. accessor, or that differentiator to the right of the method name.
        - Still need to evaluate reference questions

2021-04-22

- Attendees: josh11b, chandlerc

- Function introducer and method declaration syntax
- Resolving feedback on Generics goals proposal

2021-04-20

- Attendees: josh11b, jonmeow, gribozavr, chandlerc
- Rust `fn` problems: none in practice
- Syntax for named/keyword/labelled arguments
- Discussion about requiring all generics features to support a dynamic implementation strategy
    - Mostly came to agreement to say nice to have, but not a requirement
    - Will file a blocking issue
- Dry run of generics presentation on usage
    - Added & edited some slides on adapters

2021-04-19

- Attendees: josh11b, jonmeow, zygoloid, chandlerc
- Omitting names of function parameters
    - Require _
    - Maybe also allow _some_name, which is still discarded, but is clearer for documentation purposes
    - Or maybe <in_angle_brackets> with <> by itself allowed?
    - _ is part of the identifier lexical space, so doesn't use up
    - Maybe a keyword instead?
- Working through suggestions for "Generics Goals" proposal:
    - Discussing this line: "Generics shall provide at least as good code generation, in terms of both code size and execution speed, in all cases over C++ templates."
    - Rewrite this too: "When defining a new generic interface to replace a template, support providing using the old templated implementation temporarily as a default until types transition."
- Struct types and tuples

```
struct S {
  fn Make(Int: x) -> S;
  method (S this) Accessor() -> Int;
  method (Ptr(S) this) Mutator(Int: x);
}
var S s = S.Make(3);
if (s.Accessor() == 4) {
  s.Mutator(5);
}
var Ptr(S) p = &s;
if (p->Accessor() == 4) {
  p->Mutator(5);
```

```
}
struct T {
  var Int a;
  var Int b;
}
var T t = (.a = 3, .b = 4);
SomeFunction((.a = 3, .b = 4) as T);
```

- Can cast from a named tuple to a struct, but it is a conversion. E.g. may add fields that have defaults, change padding, etc.
- Can't cast from `Ptr(tuple)` to `Ptr(struct)`.
- Really this thing with unnamed and named members is an "argument list" not a "tuple" -- a tuple is the special case where it only has unnamed members.
- TODO(richardsmith, chandlerc): Add notes about the (limited) extent to which we think destruction order must be reverse construction order.

2021-04-13

- Attendees: josh11b, jonmeow, austern, chandlerc
- Types as values
- Proposal summaries
- `fn` vs. `func`
- Git labels vs. project
- `for (var x...)` seems like it violates the rule about looking up the syntax tree for things in scope
- Generics presentation dry run
- Allocating symbols to syntax
  - Would be nice to have a provisional syntax for generics (maybe `#`?) instead of the current placeholder (`$`)
  - Do we need to allocate a symbol to concatenation, or are we going to just use `+`?
  - Will need syntax for: error handling, generics/templates, lambda expressions, metaprogramming, ...
  - Assuming we get rid of single symbols for the preprocessor, bitwise ops, logical ops, trinary `?:` op; we have these individual symbols available: ~`!@#^|\?
  - It seems like `$` is actually available on most keyboards, at least on Mac OS X.

2021-04-05

- Attendees: chandlerc, jonmeow, josh11b
- Ways to translate `for (A; B; C)` loops with `continue`
  - Not excited by labelled break as a solution here

    - Maybe we want `while` loops that can be followed by `on continue`, `on break`, `on not break` blocks
    - Key is that `C` expression can't depend on values that are only in the body of the loop
- "Non-goal" vs. "caveat" vs. "limitation" vs. "out-of-scope"
    - Non-goal: it is a goal that we don't do this
    - Out-of-scope: not in this proposal, possibly a different proposal

2021-03-30

- Attendees: jonmeow, gribozavr, josh11b, chandlerc
- [jonmeow] Code conversion approaches
- [josh11b] Looking for generics slides feedback
- [jonmeow] Survey

2021-03-22

- Attendees: chandlerc, jonmeow, josh11b, mconst
- Considering how to present Carbon internally to Google C++ users
    - Really just exploring how to most effectively communicate and motivate things when discussing with end users as opposed to folks more deeply embedded in the language.
    - Focus on one headline feature: path to safety
    - Part of this is a "why not C++" story, answer is manifold:
        - We have tried to evolve C++ with limited success
        - More important: safety is too big a change to C++
        - C++ already has a multitude of reference types, need to simplify to make space to add safety features
        - Generics are an important part of the safety story of other languages such as Rust; don't expect templates to scale farther
        - Neither forking nor pushing the C++ committee looks cost-effective for such a big change
    - An alternative safety option is Rust, we think of Carbon as the installment-payment plan to safety; avoids large painful transition all at once
        - Carbon is aiming to provide an incremental migration and interop story with C++
        - Carbon is also designed (simpler to parse, etc.) to support tooling that enables cheaper evolution
- Planning to survey internally at Google re: C++

```
// Newest model supports this:
fn F[A:$ T, B(.L = T.X.Y.Z):$ U](...) { ... }
// even though it forbids LexBroken below.
```

- How bad do we think the Swift undecidable system is anyway?

    - Literature on Knuth-Bendix algorithm complexity includes things like this simple non-terminating_example, and this thesis which talks about runtimes in the seconds to hours for examples with 10s of rules
    - Interesting background from Slava: https://forums.swift.org/t/formalizing-swift-generics-as-a-term-rewriting-system/45175/9
- My decidable constraint system might also reject the conditional conformance turing machines in Rust: https://sdleffler.github.io/RustTypeSystemTuringComplete/

2021-03-16

- Attendees: dabrahams, jonmeow, josh11b, zygoloid, gribozavr
- [josh11b, jonmeow, zygoloid] What goes in a "principle" vs. "goals of a specific feature" doc?
- [dabrahams, jonmeow] Improving the readability of the executable semantics code
- [josh11b, dabrahams, zygoloid, gribozavr] Thinking about ordering was helpful for generic type equality algorithm.

```
interface B {
  var Type:$ X;
}
interface A {
  var Type:$ T;
  var B:$ U where U.X == T;
}
interface C {
  var A:$ V where V.T == V.U;
}

A:V
  B:V.U
  V.U.X == V.T
V.T == V.U

V.T -> V.U
V.U.X == V.U

interface LexBroken {
  var LexBroken:$ A;
  var LexBroken(.B = A.A.B.B):$ B;
}

fn F[LexBroken:$ T](T: x) {
```

```
  // T.B.B not canonical
  // == T.A.A.B.B not canonical
  // == T.A.A.A.A.B.B not canonical
  // etc.
}

Fix is to say, primary ordering is by number of dots

interface Broken {
  var Broken:$ A;
  var Broken(.B = A.A.B.B):$ B;
  var Broken(.C = A.A.B.B):$ C;
}

fn F[Broken:$ T](T: x) {
  // T.C.C == T.A.A.B.B == T.B.B
}

// Possible solutions:
   -  if there is more than one choice when applying rewrites, require user
      to put in explicit casts/coercions to specify an explicit path that
      proves the types are equal -- concern it won't allow evolution
   -  the compiler will only perform one step, require user to do casts if
      you need more than one; if you have a common equivalence, add it as a
      redundant constraint in the interface so it can be done in one step.

Concern: What if we have equality between two types with different type
bounds? Motivates wanting to explicitly cast to get a particular API.

struct LB {}
impl LexBroken for LB {
  var LexBroken:$ A = LB;
  var LexBroken:$ B = LB;
  // can we prove B == A.A.B.B? yes
}

One possibility is only allow at most one . to the right of the =, and no
forward references
Another possibility is no dots to the right of the =, but allow forward
references
When you need to introduce something that doesn't fit, introduce a new
variable
```

2021-03-15

- Attendees: josh11b, dabrahams, chandlerc, zygoloid
- Two big tasks in generics:
    - How much expressivity can we have in constraints while keeping an efficient type equality decision procedure.
    - How much expressivity is needed in practice for constraints, for example what is used by the Swift standard library.
- Do we need templates, or do we think we can get away with just generics?
    - Chandler: Design for templates, so that the design accommodates them even if we don't end up including them in Carbon.

```
// Expressivity constraint in an interface
interface Foo {
  // Some associated types
  var ...:$ A;
  var ...:$ B;
  // X is some single id, a member of Z
  // Y is an expression, starting with A, B, or C, or `Self`.
  // If Y starts with `C`, can have at most one `.`
  // X may appear at most once on the left of the == in this list of where
clauses.
  var Z:$ C where C.X == Y;
  // equivalent to var Z(.X = Y):$ C
  // Allows: C.X == C
  // Allows: C.X == C.W
  // Allows: C.X == A
  // Allows: C.X == A.W.T
  // Allows: C.X == Self
  // Does not allow forward reference: C.X == D.W
  // Does not allow: C == A
  // Does not allow: C.X.W == A
  var ...:$ D;
}

fn F[Foo:$ T](...)
// T canonical in the context of F.
// No constraints on T beyond being Foo (since no `where` on `F`)
// implies T.A, T.B, T.C, T.D are all canonical
// => means that these are all in different equivalence classes
// where C.X == anything
// implies C.X is not canonical, its canonical representative is
// found from evaluating `anything`
// `anything`'s canonical can be determined due to no forward reference
// Exception is dealing with C.X == C.W. In this case, C.X is not canonical
// eliminate C.X, use C.W in its place. If C.W == C.X appears later, it is
```

```
// redundant so drop it.
C.X == C.W, C.W == A, C.? == C.X

What about T.C.U? Have to look at Z, and in particular the constraints that
involve Z.X.
We already established X is not canonical, so X.anything is not canonical.
So any constraint in Z on X
  var ...:$ X where X.? == V;
implies V is not canonical
fn F[Foo:$ T](...) where T.A == Int;

The interface has implications in terms of type equality
These implications create equivalence classes
The canonical type is a chosen representative for its equivalence class

Use different naming for types vs. interfaces



interface A {
  var ...:$ X;
  var ...:$ Y where Y.Z == X;
}
interface B {
  var A:$ Q where Q.Y == Q.X.W;
}
In B, have Q.Y.Z == Q.X (from A), but also Q.Y == Q.X.W. OK?

fn F[B:$ T]
T is canonical
T.Q is canonical
T.Q.Y is not canonical = T.Q.X.W may or may not be canonical
evaluate T.Q.X.W
T.Q.X not canonical, it is T.Q.Y.Z == T.Q.X.W.Z

T.Q.X.W.Z.W.Z
```

2021-03-08

- Attendees: josh11b, mconst, dabrahams
- Constraints for generics

These two are equivalent in Rust:

```
F<T>(x: T) where T: Comparable
F<T: Comparable>(x: T)
```

and in Swift:

```
func f<T>(x: T) where T: Comparable {}
func f<T: Comparable>(x: T) {}
```

Interface with associated type:

```
interface Container {
  var Type:$ Elt;
}

fn F1[Container:$ T](T: x) where T.Elt == Int;
fn F2[Container(.Elt = Int):$ T](T: x);

fn G1[Container:$ T, Container:$ U](T: x, U: y) where T.Elt == U.Elt;
fn G2[Type:$ E, Container(.Elt = E):$ T, Container(.Elt = E):$ U](T: x, U:
y);

// from https://forums.swift.org/t/swift-type-checking-is-undecidable/39024
protocol Impossible {
  associatedtype A : Impossible
  associatedtype B : Impossible
  associatedtype C : Impossible
  associatedtype D : Impossible
  associatedtype E : Impossible
    where A.C == C.A
          A.D == D.A
          B.C == C.B
          B.D == D.B
          C.E == E.C.A
          D.E == E.D.B
          C.C.A == C.C.A.E
}

protocol Impossible {
  associatedtype A(.C = AC_CA, .D == AD_DA) : Impossible
  associatedtype B(.C = BC_CB, .D == BD_DB) : Impossible
  associatedtype C : Impossible
  associatedtype D : Impossible
  associatedtype E : Impossible
  associatedtype AC_CA where AC_CA == A.C, AC_CA == C.A;
  associatedtype AD_DA where AD_DA == A.D, AD_DA == D.A;
  associatedtype BC_CB where BC_CB == B.C, BC_CB == C.B;
  associatedtype BD_DB where BD_DB == B.D, BD_DB == D.B;
```

```
  associatedtype EC = E.C
  associatedtype ED = E.D
  associatedtype CC = C.C
  associatedtype CCA = CC.A

  associatedtype CE_ECA where CE_ECA == C.E, CE_ACA == EC.A;
  associatedtype DE_EDB where DE_EDB == D.E, DE_EDB == ED.B;
  associatedtype CCA_CCAE where CCA_CCAE == CC.A, CCA_CCAE == CCA.E;
}

//
https://docs.google.com/document/d/1iu7o_FxWT2Lz4cWkKUgCsA58I1E0e6Jry7g7wacX
TsU/edit
interface CImp {
  // for A.C == C.A
  var CImp:$ AC_CA;

  // for C.C.A == C.C.A.E
  var CImp:$ CCA_CCAE;

  // associatedtype A : CImp where A.C == AC_CA;
  var CImp(.C = AC_CA):$ A;

  var CImp(.A = AC_CA, .AC_CA = CCA_CCAE):$ C;
}

fn X[A:$ B, C:$ D, E(.F = B, .G = D):$ H](...) {
  is B.X.Y == H.Z.W?
  want every expression is either canonical or a pointer to something that
is transively canonical
  B, D, H are canonical
  require: nothing about B's or D's members being canonical can be affect by
being used as an argument to E
  Let's look at E

  interface E {
    var ... :$ F;
    var ... :$ G;
    var ... :$ Z;
  }
  H.F not canonical == B canonical
  H.G not canonical == D canonical
  Either E.F's bounds involve E.Z or not.
  If neither E.F nor E.G's bounds involve Z, then Z is canonical
  Else Z will have the came canonical type as an expression F.something or
G.something
```

```
  interface E {
    var J(.W = Z):$ F;
    var J(.W = Z):$ G;
    var ... :$ Z;
  }
  in this case, it would illegal to write
  E(.F = B, .G = D):$ H
  unless
  B.W == D.W

[...:$ K, A(.W = K):$ B, C(.W = K):$ D, E(.F = B, .G = D):$ H]

struct HH(.F , .G) {
  impl E(.F = ..., .G = ...) { ... }
}

fn Q[...:$ K, A(.W = K):$ B, C(.W = K):$ D](B: b, D: d) {
  var HH(.F = B, .G = D): e;
  X(b, d, e);
}

protocol P {
  associatedtype A: Q
}

protocol Q {
  associatedtype B: P
}

fn R1[Container:$ T](T: x) where T.SliceType == T;
// R2 requires something like `letrec`, or the order
fn R2[Container(.SliceType = T):$ T](T: x)

// Similarly
interface HasAbs {
  var Type:$ Magnitude;
  method (Self: this) Abs() -> Magnitude;
}
fn UseAbs[HasAbs(.Magnitude = T):$ T](T: x) -> T {
  return x + x.Abs();
}

// Totally unclear if this could work with some keyword-ish-like `.Self`
// placeholder? Not quite the same as `Self` in the definition (hence
// starts with `.` to reflect it coming from within the interface), but
```

```
// maybe more constrained than full generality of letrec.
fn UseAbs[HasAbs(.Magnitude = .Self):$ T](T: x) -> T {
  return x + x.Abs();
}

interface Container {
  var Container(.SliceType = .Self):$ SliceType;
  method (Self: this) Clone() -> Self;
}


interface Impossible {
  //     AC_CA.AC_CA == AC_CA.CAE
  var Impossible:$ AC_CA_CAE;
  var Impossible(.AC_CA = AC_CA_CAE, .CAE = AC_CA_CAE):$ AC_CA;

  var Impossible:$ AD_DA;

  //     BC_CB.AC_CA == BC_CB.CAE
  var Impossible:$ BC_CB_CAE;
  var Impossible(.AC_CA = BC_CB_CAE, .CAE = BC_CB_CAE):$ BC_CB;

  var Impossible:$ BD_DB;

  //     CE_ECA.AC_CA == CE_ECA.CAE
  var Impossible:$ CE_ECA_CAE;
  var Impossible(.AC_CA = CE_ECA_CAE, .CAE = CE_ECA_CAE):$ CE_ECA;

  var Impossible:$ DE_EDB;

  //     CCA_CCAE.AC_CA == CCA_CCAE.CAE
  var Impossible:$ CCA_CCAE_CAE;
  var Impossible(.AC_CA = CCA_CCAE_CAE, .CAE = CCA_CCAE_CAE):$ CCA_CCAE;

  var Impossible:$ CAE;

  var Impossible:$ AE;

  var Impossible(.C = AC_CA, .D = AD_DA, .E = AE):$ A;
  var Impossible(.C = BC_CB, .D = BD_DB):$ B;
  var Impossible(.A = AC_CA, .B = BC_CB, .E = CE_ECA,
                 .AC_CA = CCA_CCAE, .CAE = CCA_CCAE, .AE = CAE):$ C;
  var Impossible(.A = AD_DA, .B = BD_DB, .E = DE_EDB):$ D;
  var Impossible(.AC_CA = CE_ECA, .BD_DB = DE_EDB):$ E;
}
```

2021-03-02

- Attendees: jonmeow, josh11b, zygoloid, chandlerc
- [jonmeow, josh11b] for loops and ranges (`for x in 0..10`)
    - [josh11b] Don't want , operator
- [zygoloid] `in` vs `:` — Swift and Rust use in without parens, maybe different? (Kotlin has parens).
- [josh11b] Discord polls?
    - [josh11b] Maybe a good way to get quick feedback
    - [zygoloid] Easy to miss, people change opinions
    - [jonmeow] Usually only 4-5 votes, not too useful
- [josh11b] Make var an expression? Also discussing how to assign a tuple to two separate identifiers.
    - chandlerc joined during this
- [jonmeow] Back to for ;; to ask chandlerc
    - More general construct for loops that can for ;;
- (chandlerc leaves)
- [josh11b] generics questions for zygoloid

2021-03-01

- Attendees: jonmeow, josh11b, mconst
- [jonmeow, josh11b] `:` vs not in var syntax, for consistency with pattern matching
- [mconst, josh11b] deducible vs. multi interface parameters
    - Question: Is using associated types for all deducible arguments acceptable?

```
interface Stack {
  var Type:$ Element;
  method (Ptr(Self): this) Push(Element: value);
  method (Ptr(Self): this) Pop() -> Element;
  method (Ptr(Self): this) IsEmpty() -> Bool;
}

fn PeekAtTopOfStack[Stack:$ StackType](Ptr(StackType): s)
    -> StackType.Element {
  var StackType.Element: top = s->Pop();
  s->Push(top);
  return top;
}

fn SumIntStack[Stack(.Element = Int):$ T](Ptr(T): s) -> Int {
  var Int: sum = 0;
  while (!s->IsEmpty()) {
```

```
    sum += s->Pop();
  }
  return sum;
}

interface Iterator {
  var Type:$ Element;
  method (Ptr(Self): this) Advance();
  method (Ptr(Self): this) Done() -> Bool;
}
interface Container {
  var Type:$ Element;
  var Iterator(.Element = Element): IterT;
  // Sad forward reference:
  // var Container(.SliceType = SliceType): SliceType;
  // Except `Self` means something else already here:
  // var Container(.SliceType = Self): SliceType;
  // Maybe a new keyword?
  // var Container(.SliceType = recursive): SliceType;
  // var FixedPoint(lambda S => Container(.SliceType = S)): SliceType;
  var Container: SliceType;
  requires SliceType.SliceType == SliceType;

  // Can we forward declare SliceType?
  var Container: SliceType;
  override Container(.SliceType = SliceType): SliceType;

  method (Ptr(Self): this) Begin() -> IterT;
  method (Ptr(Self): this) Slice(IterT: start, IterT: end) -> SliceType;
  ...
}


fn SortContainer[Comparable:$ Element,
                 Container(.Element = Element):$ ContainerType]
    (Ptr(ContainerType): container_to_sort);

fn EqualContainers[HasEquality:$ ET,
                   Container(.Element = ET):$ CT1,
                   Container(.Element = ET):$ CT2]
    (Ptr(CT1): c1, Ptr(CT2): c2) -> Bool;

interface MapInterface {
  var EqualityComparable:$ Key;
  var Type:$ Value;
  method ...;
```

```
    impl Container(.Element = (Key, Value));
    alias IterT = Container.IterT;
  }

  // Alternatively, using `extend` which means `impl` & `alias` all the names
  interface MapInterface {
    var EqualityComparable:$ Key;
    var Type:$ Value;
    method ...;
    extends Container(.Element = (Key, Value));
  }

  interface RandomAccessIterator {
    ...
    extends Iterator;
  }

  interface RandomAccessContainer {
    var Type:$ Element;
    var RandomAccessIterator(.Element = Element):$ IterT;
    extends Container(.Element = Element, .IterT = IterT);
    // Sad: name collisions and covariance issues.
  }

  interface RandomAccessContainer {
    extends Container;
    requires RandomAccessIterator(.Element = Element): IterT;
  }

  interface RandomAccessContainer {
    extends Container;
    // Redefinition allowed by covariance.
    override RandomAccessIterator(.Element = Element):$ IterT;
  }

  interface Pair {
    var Type:$ Left;
    var Type:$ Right;
  }
  structural interface SamePair {
    // Doesn't work: we don't want to require types to have a `T`.
    var Type:$ T;
    extends Pair(.Left = T, .Right = T);
  }
  structural interface SamePair {
    extends Pair(.Left = .Right);
```

```
}
structural interface SamePair {
  extends Pair;
  requires Left == Right;
}

fn F(DynPtr(MapInterface where .Key == .Value): m);
fn F(DynPtr(MapInterface): m where m.Value.Key == m.Value.Value);
fn F(DynPtr(structural interface { extends MapInterface; requires Key ==
Value; }): m);

// Unfortunately, this references Container, and so can't be used in the
// definition of Container.
structural interface SelfSliceContainer {
  extends Container;
  requires SliceType == Self;
}

struct Hashmap(EqualityComparable + Hashable:$ K, Type:$ V) {
  impl MapInterface {
    // Covariance!
    var EqualityComparable + Hashable:$ Key = K;
    var Type:$ Value = V;
    ...
  }
}

struct Hashmap(EqualityComparable + Hashable:$ K, Type:$ V) {
  impl MapInterface {
    // Covariance!
    var EqualityComparable:$ Key = K;
    var Type:$ Value = V;
    ...
  }
  // Redefinition is allowed by covariance.
  override EqualityComparable + Hashable:$ Key = K;

  impl Container {
    var ...: SliceType = ...;
  }
}


struct Treemap(EqualityComparable + Comparable:$ K, Type:$ V) {
  impl MapInterface {
    // Covariance!
```

```
      var EqualityComparable + Comparable:$ Key = K;
      var Type:$ Value = V;
      ...
    }
  }


  ...: B, ...: C, A(B, C): D, ...

  is B.X == D.Y

  st
```

2021-02-23

- Attendees: jonmeow, josh11b, dabrahams, chandlerc
- Not much discussion for first hour, until chandlerc joined (after dabrahams left)
- [chandlerc] Google project discussion

2021-02-22

- Attendees: chandlerc, jonmeow, josh11b, geoffromer, dabrahams
- [chandlerc, jonmeow] Alternative core team structure and review ideas
- [chandlerc, dabrahams] IDE code completion, etc and language constraints
- [chandlerc, dabrahams, geoffromer] Long discussion about abstraction boundaries interacting with loops and some of the challenges
    - especially -- could we use a coroutine oriented model for loops? would it lower with too much overhead? Could we avoid that?
        - Why? https://en.wikipedia.org/wiki/Sather#Example_of_iterators
    - C++ ranges/generators seem to maybe not to have succeeded
    - [chandlerc] big thing is to bound/constrain amount of code inside abstraction boundaries (inside body of co-routine, etc).
    - nice thing about corountines is that it makes it easier to express the code for iteration
        - definitely true in complex cases
        - maybe not in simple cases?
    - maybe a happier place is similar to Python's model (and likely other models) where there is a simpler iteration abstraction that loops use, and then let coroutines be used to implement those where useful?
        - Big question is whether loops *presume* coroutines, or just an option for handling the complex cases.
        -

2021-02-18

- No notes

2021-02-16

- Attendees: jonmeow, dmitrig, chandlerc, dabrahams, zygoloid, geoffromer
- [chandlerc] What to call Clang's 'sema' (semantic analysis) equivalent
    - May move code into toolchain, call it semantics
- [dabrahams] consensus in decisions
- [chandlerc, geoffromer] provisionality and placeholders

2021-02-11

- No notes

2021-02-09

- Attendees: jonmeow, zygoloid, chandlerc, dabrahams, josh11b, mmdriley, gribozavr
- [zygoloid, chandlerc, dabrahams, josh11b] String literals
- [mmdriley, chandlerc, jonmeow] Proposal process
- [chandlerc, jonmeow, mmdriley, dabrahams] GitHub comment flow
- [jonmeow] NOTE: discussion continued, but I don't think participants took notes

2021-02-08

- Attendees: josh11b, mconst, jonmeow, chandlerc, mmdriley, dabrahams
- [josh11b, mconst] Generics (jonmeow didn't really catch context)
- [chandlerc, mconst, jonmeow, mmdriley] Toolchain bootstrap changes
    - Is the bootstrap build crazy?
    - Probably a reasonable solution for now, can start looking at providing more if needed
- [chandlerc, mmdriley] Windows port/support (trying to set up VM)
- [mconst, chandlerc] Language discussion with Rust context
- [dabrahams, chandlerc, mconst] defer vs. RAII
    -
- [dabrahams, chandlerc, mconst] discussing motivations around consume-semantics and immutable view semanics with parameters
- [dabrahams, chandlerc] definitive initialization vs. unformed state

2021-02-04

- Attendees: josh11b, mmdriley, jonmeow
- [josh11b, mmdriley] A bit about the josh11b & mconst's errors proposal
- [josh11b, mmdriley] What has happened to Carbon in the last year?
- [josh11b, mmdriley] What is going on with Safety in Carbon?
- [jonmeow, josh11b] Figuring out how to act on comments on the safety syntax proposal.

2021-02-02

- Attendees: josh11b, jonmeow, dabrahams, chandlerc
- [jonmeow] Safety decision, get suggested edits from chandlerc
- [jonmeow] Discussion process, how to get feedback from core team members
    - Discussion leaning towards less formal feedback-based core team meetings
    - Discussion threaded onto proposal complexity
- [dabrahams] Difficulty of setting up tooling
- [josh11b] Swift raw strings
    - [dabrahams] Assume the main use is regular expressions, haven't heard complaints
    - [gribozavr] Similar

2021-02-01

- Attendees: jonmeow, zygoloid, josh11b, chandlerc
- [josh11b] Discussing comments on comments PR
    - When we have an experimental part of a proposal, do we expect that part to remain as-is, or do we expect it to be revised? Should we avoid building further proposals on top of experimental decisions? (Examples: block comments, build modes)
    - Threaded off on discussion about build modes and the expected number
- [zygoloid] Raw string literals
    - Discussing Swift's use (#"foo\#nfoo"# is equivalent to "foo\nfoo")
    - May switch to Swift-style raw string literals instead of Rust, need to talk with people familiar with Swift
        - Advantage: raw strings are a generalization of regular strings rather than a separate feature
        - Advantage: can use `\#` to escape a final newline, `\#n\#` to include trailing whitespace, `\#t` to include tabs, etc. in a raw block string literal – no loss of functionality when going from regular to raw string literals
        - Disadvantage: "\\######" etc become less convenient to express. Such strings quite rare but do exist in practice.

- [chandlerc] Likes https://chrome.google.com/webstore/detail/meet-shortcuts/gkppodhmelnfemfhnbacnbobemdjaojj (keyboard shortcuts for Google Meet)
    - Some discussion about ergonomic keyboards
- [josh11b] Sum types: starting to more strongly prefer type-indexed approach

2021-01-28

- Attendees: jonmeow

2021-01-26

- Attendees: jonmeow

2021-01-25

- Attendees: jonmeow, chandlerc, josh11b, mconst
- [jonmeow] Discussing executable semantics and getting things to compile
    - [chandlerc] Will give advice on tool PR
- [josh11b] Discussing error draft with chandlerc
    - Note: claims to represent MConst's opinion without him having reviewed the document
    - (mconst later joined discussion)
- [jonmeow] Minor side-question on `using std::`
    - Branched into discussion of alias
    - Use of Carbon vs a name-agnostic Std vs something clearer than Std for the standard library
- [chandlerc,mconst] Implicit parameters
    - Logging
    - Memory allocator
    - Effects that you want to fake out during tests
    - Floating point context
        - Could inherit the caller's context, or could enforce that we are in the mode this function requires.
        - Overload the function based on the current floating point rounding mode?
    - Question: is this represented explicitly in the function's signature?
        - Capture the potentially visible side effects of the function
        - Possibly propagated incrementally by using default arguments?
        - Possibly can say "everything in this package uses this implicit argument"? Analogous to imports
    - Performance costs?
        - Actually implemented using TLS or globals
        - Static in release mode, dynamic in test mode?
        - Optimize using whole program analysis?

2021-01-21

- Attendees: gribozavr, josh11b
- Read "The Next 7000 Programing Languages"
  - Early sections, and plant analogy, less valuable than later sections.
  - Doesn't capture why it is hard to evolve JavaScrip
- Read "Flow-Sensitive Type Qualifiers"
  - Reminiscent of the state tracking re: uninitialized/unformed values, and the legal operations you can perform depending on that state.
  - Does this specify an algorithm that is sufficiently deterministic that a language spec could use it to determine whether something is known in a flow-sensitive way at a given point in the program? In particular, could you rely on the result of this algorithm to be the same across versions and vendors of compilers?
  - Would this be a valuable thing for users to specify when defining their own types? Like "type state" considered and rejected by Swift.
  - Could this be used to annotate that iterators become invalidated when their container is modified?
  - Josh likes that this feels a bit more optional than types, and maybe could be the basis of a gradual system for adding more proofs of correctness to code. Being optional is good for accommodating different users, and allowing graceful transition when some code is very dynamic or hard to prove properties of.

2021-01-19

- Attendees: chandlerc, gribozavr, jonmeow
- Discussing There's plenty of room at the Top: What will drive computer performance after Moore's law? on software vs CPU performance
  - Has successfully gotten some attention at Google and is expected to help Carbon's efforts

2021-01-14

- Attendees: chandlerc, gribozavr, josh11b, zygoloid, geoffromer
- [chandlerc, gribozavr, josh11b, zygoloid, geoffromer] Sum types

Observation: some of the following refer to Optional constructors (probably at least #2), some refer to match functions (at least #three and #four).

```
var Optional(Int): x = ...
match (x) {
  case .Some(1) => ...
  case Optional(Int).Some(2) => ...
  case .Some(Int: three) => ...
```

```
  case Optional(Int).Some(Int: four) => ...
}

var Optional(Optional(Int)): y = …
match (y) {
  case .Some(.Some(5)) => …
  case .Some(Optional(Int).Some(6)) => …
}


fn Foo() -> Optional(Optional(Int)) {
  return .Some(.Some(7));
}
.Bar(8) == (.Bar = 8,) or (.Bar = (8,),)
has type (.Bar = Int,) or (.Bar = (Int,),)


var auto: x = .Foo(8);
var Optional(Int): y = .Somw(3); // error, can't convert
(.Somw = Int,) to Optional(Int)

var Variant(Int, String): q = …
match (q) {
  case .Value(Int: i) => { … }
  case .Value(String: s) => { … }
}

interface MatchContinuation {
  var Type:$$ ReturnType;
  fn Value(Int: i) -> ReturnType;
  fn Value(String: s) -> ReturnType;
}


choice Expected(Type: T) { Success(T: _), Failed(Error: _) }

var Variant(Expected(Int), Expected(String), Optional(Int)): v =
…
match (v) {
  case .Value(.Failed(Error: e)) => … // ambiguous?
  case .Value(.Some(Int: n)) => … // ok?
  case .Value(Expected(Int).Failed(Error: e)) => … // ok?
}
```

```
Match(MatchContinuation:$$ T: x)
Match(fnty(Int, Int): f)
match (v) { case (a, b) => g() }
v.Match(lambda (a, b) { g() } }

// chandlerc's nicer syntax if we can
let Optional(Int):$$ w = Optional(Int).Some(4);
match (v) {
  case Expected(Int).Failed(Error: e) => …
  case Optional(Int).Some(3) => …
  case w => …
  case Optional(Int).Some(Int: n) => …
}

var (Optional(Int), Optional(String)) p = ...;
match (p) {
  case (.Some(Int: n), .Some(String: s)) => ...  // forbidden?
}

match (v) {
  case Expected(Int): .Failed(Error: e) => …
}


// totally unrelated musings
match (v) {
  case Expected(Int).Failed(Error: e) => ...
  case (is Optional(Int).Some(3)): x => ...
  case == Optional(Int).Some(3) => ...
  case is w => ...
  case Optional(Int) => ...
}

var Expected(Int).Failed(Error: e): f = Foo();  // OK?
Expected(Int).Failed(Error)

(Variant(...), Int)
case (Optional(Int), is 3))

// Match to dynamically determine which subtype
class B { ... }
class D1 extends B { ... }
class D2 extends B { ... }
var Ptr(B): p = ...;
match (*p) {
  case D1 => ...;
```

```
  case D2 => ...;
}

var Variant(Optional(Int), Optional(String)): v = …;
match (v) {
  case .Value(.Some(Int: n) as Optional(Int)) =>
  case .Value(.None as Optional(Int)) =>
```

Proposed idea: dot changes the direction of type inference for the *name* that follows the dot. And this is just like ordinary member-access usages of dot.

Maybe require sum type to implement its own MatchContinuation interface, as minimal check of bidirectionality?

Maybe have explicit syntax for requesting comparison by ==?

Options:
- Patterns and expressions have same grammar, some functions can go both ways
- Patterns are superset of expressions (e.g. leading dot), things not in subset are matched with ==
- Patterns and expressions have separate syntax

If leading-dot is how we request matching rather than equality comparison, we need some type-assertion/disambiguation syntax.

2021-01-12

- Attendees: jonmeow, gribozavr, josh11b, zygoloid
- Some discussion of Swift
- Discussing roadmap, adding comments
- Discussing zygoloid's comments on safety principles, arithmetic/overflow
- Sum types:
    - Proxy approach: simple, bounded, easy to understand, but incomplete (still need a primitive notion of matching for sufficiently simple values)
    - Callback approach: elegant (does not rely on another construct for foundation), powerful (but maybe more power than is good?), but more opportunity to do something outside the expected pattern leading to surprising behavior. Example concern: what happens if two callbacks are called?
    - Compositional concern: nested matches? Both work but with some awkwardness.
        - Going to end up with a lot of callback functions for the callback approach, when dealing with tricky cases like nested patterns.

- Interesting case: N-tuple of optionals. Does this lead to 2^N callbacks? In the proxy approach, the compiler has visibility into the decision criteria.
- Compiler has some freedom to generate proxies, shouldn't be expected to have side effects.
- With callbacks, more potential for lifetimes and side effects in the match operator function. More observable when the compiler calls the user's function.
- ```
  match x {
    case (0, 0, _) => …
    case (0, 1, _) => …
    case (1, 0, _) => …
    case (1, 1, .Some(5)) => …
    case (1, 1, _) => …
  }
  ```
  Is it OK to call the match function for the third element of the tuple even if we're not in the 4th / 5th case? (Can we directly build a single jump table?)
  - Richard thinks: maybe yes in the proxy approach, maybe no in the continuation approach
  - Josh thinks this should be left up to the implementation in either approach
- ```
  match (x) {
    // imagine match function unpacks, then calls callback with
    // mutable references to unpacked elements, then repacks
    case v@.PackedPair(0, 0) => { v = .PackedPair(1, 1); }
  }
  ```
- ```
  // Concern about updating the value after the case function returns
  // overwriting another change to the value made by the user.
  match (x) {
    case .Some(_) => { x = .None; }
    case .None => { x = .Some(-1); }
  }
  // Suggests it might be too dangerous to mutate *this
  // in a match operator.
  ```
- ```
  // callback approach easily supports lifetimes of temporaries that
  // would need to live for the whole case body.
  method (Ptr(Self): this) operator match((ChoicesInterface:$ T): callback) {
    var InnerType: temporary = this->unpacked_from_rep();
    callback->Option(SomeType(&temporary));
    // temporary gets destroyed and deallocated after callback returns
  }
  // Similar thing in the proxy approach via repeated calls to operator
  ```

```
match
// All operator match() return values would need to remain alive for
the
// entirety of the match expression.
private struct Unpacked {
  var InnerType: temporary;
  method (Ptr(Self): this) operator match() -> SomeType {
    return SomeType(&this->temporary);
  }
}
method (Ptr(Self): this) operator match() -> Unpacked {
  return Unpacked(this->unpacked_from_rep());
}
```

- Proxy approach is the initial algebra corresponding to the F-algebra from the continuation approach. (https://www.schoolofhaskell.com/user/bartosz/understanding-algebras)
- ```
  var Optional(Int): x = …
  match (x) {
    // These two '.Some's mean completely different
  things?
    case .Some(3) => … // decompose and compare int
    case Optional(Int).Some(3) => … // compare
  Optional(Int)
  }
  ```
- ```
  var Byte: y = ... ;
  match (y) {
    case in 0..127 => ...
    case in 128..255 => ...
  }
  // Compare to:
  struct Has3PossibleValues {
    private var Byte: z;
    fn One() -> Self;
    fn Two() -> Self;
    fn Three() -> Self;
  }
  impl EqualityComparable for Has3PossibleValues;
  var Has3PossibleValues: w = ...;
  match (w) {
    // Can we possibly have exhaustive cases?
    // Compiler can't tell if there are mutating methods
    // that would lead to other values, random numbers,
  etc.
    case Has3PossibleValues.One() => ...
    case Has3PossibleValues.Two() => ...
  ```

```
  case Has3PossibleValues.Three() => ...
}
```

- Aside: `0..127 == Std.Range(0, 127)`
  `Array.Operator[]` is overloaded to take a range and return a slice
- Question: Does `'case 0..127' work by supporting a general Int == Range(Int) comparison?`
  Would you rather write:

```
if (7 == 0..127) { ... }
```

  or

```
if (7 in 0..127) { ... }  // definitely this is better
```

  ?

```
case Has3PossibleValues(in 1..3) =>
```

  Idea: `in expr` is a pattern that matches if `scrutinee in expr` evaluates to `True`, just like `expr` is a pattern that matches if `scrutinee == expr` evaluates to `True`

```
p^in 1..3
a * (*b + 1), *(*b)
```

2021-01-11

- Attendees: josh11b, zygoloid
- Talked about sum types design direction proposal.
  - With inlining, callback approach and pattern matching proxies are probably going to produce equivalent code. Without inlining the callback approach is probably going to be harder to optimize (contrary to what is suggested by the doc at the moment).
  - Lifetime issues probably should be included in the proposal, for two reasons: trying to produce a value with a pointer to a temporary is a case, and also giving a chance for modifications to the object made during the `case` block to be propagated back to the storage representation, when we allow mutations.
  - Summarized in discord.
- Briefly mentioned recent changes to Josh's constructing derived types doc.

2021-01-07

- Attendees: josh11b, jonmeow, zygoloid
- (josh11b joins)
- (jonmeow joins)
- A little discussion about Google-internal work
- Brief discussion of ongoing proposals
  - chandlerc's initialization idea
    - josh11b is trying to figure out how this would work with inheritance, see doc
  - jonmeow's safety principle and syntax guidance RFCs
    - josh11b means to review safety principle

        - jonmeow is working on syntax guidance
        - Some discussion about safety principles and related options
            - Discussing trade-offs of compile-time safety
            - josh11b recalls an algorithm that used uninitialized memory access for better performance — https://research.swtch.com/sparse
- (zygoloid joins)
- josh11b made a doc about initialization in inheritance
    - Most concerned about having private base members & C++ interop
    - Discussing trade-offs at length