Still To Come



Kate Gregory

@ gregcons | www.gregcons.com/kateblog

Generics



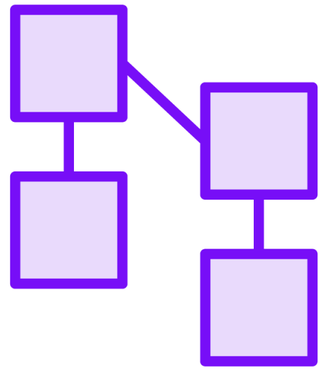
Like C++ templates, but checked at compile time



Defined in terms of interfaces



Compiler checks that calling code is using a type that implements the interface



Compiler checks the generic only uses capabilities from the interface

```
fn SortVector(T:! Comparable,
              a: Vector(T)*) { ... }

// iv is a Vector of i32
SortVector(i32, &iv);

// sv is a Vector of String
SortVector(String, &sv)

fn SortVectorDeduced[T:! Comparable]
       (a: Vector(T)*) { ... }

SortVectorDeduced(&iv);

SortVectorDeduced(&sv);

interface Comparable {
   fn Less[self: Self](rhs: Self)
       -> bool;
}
```

- Can sort a vector of anything
- You tell it what the vector holds

- Using a deduced parameter makes the calls easier and prevents bugs

- Compiler deduces T is i32

- Compiler deduces T is String

- The interface specifies the methods T must implement

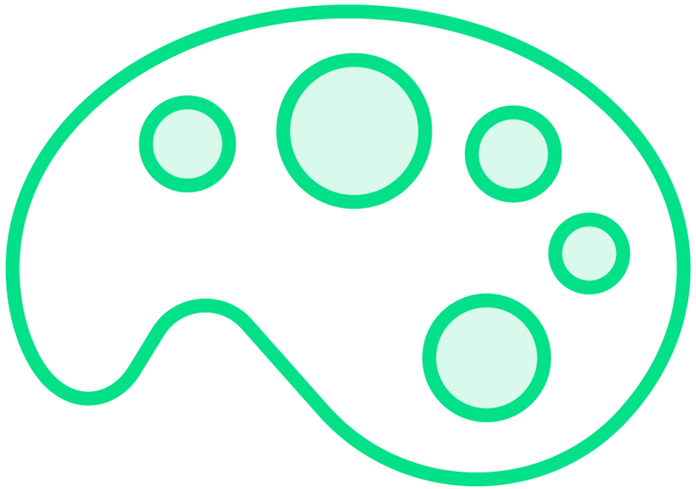
C++ Interop

Use C++ types and functions in Carbon code

Use Carbon types and functions in C++ code

Not mixing and matching languages in one code file

Variadics



Takes the idea of Generics further

- SortVector works on a vector of anything
- But it’s any one thing

A variadic function takes any number of arguments

- Each of which can be any type
- Or implement interfaces

Variadic functions are as old as C

- printf

Variadic types in C++ are somewhat newer

- tuple

```
fn SumInts(... each param: i64)
   -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- Takes a pack of i64 elements (any number)
- Returns an i64
- Initializes the return value
- Will be expanded into as many += as there are elements in the pack

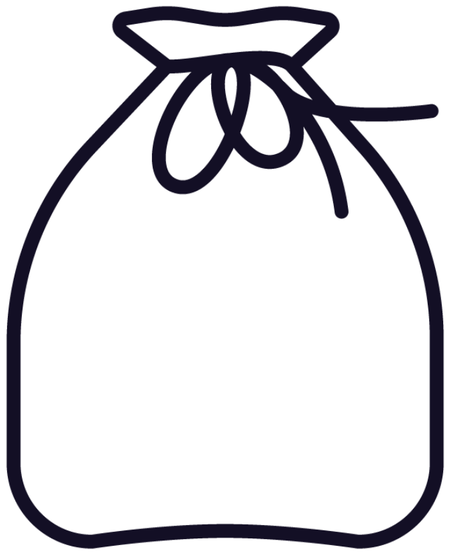
```
fn Min[T:! Comparable & Value]
   (first: T, ... each next: T)
   -> T {

  var result: T = first;
  ... if (each next < result) {
     result = each next;
  }
  return result;
}
```

- Takes a pack of elements, all the same type
- The first element is declared separately

- This will produce as many ifs as there are elements in next

Tuples



Tuples are built into Carbon

- Not a library type

Declare with parentheses:

```
let Nums: (i32, i32, i32) = (1, 2, 3);
```

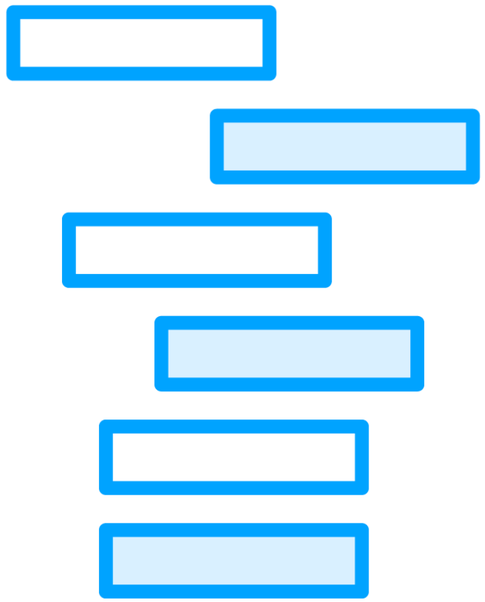
Access with . and an index

```
return nums.0 + nums.1;
```

Functions can return tuples

```
fn DoubleBoth(x: i32, y: i32)
  -> (i32, i32) {
    return (2 * x, 2 * y);
  }
```

Variadic Example: Zip



Combine any number of vectors of different types into a vector of tuples

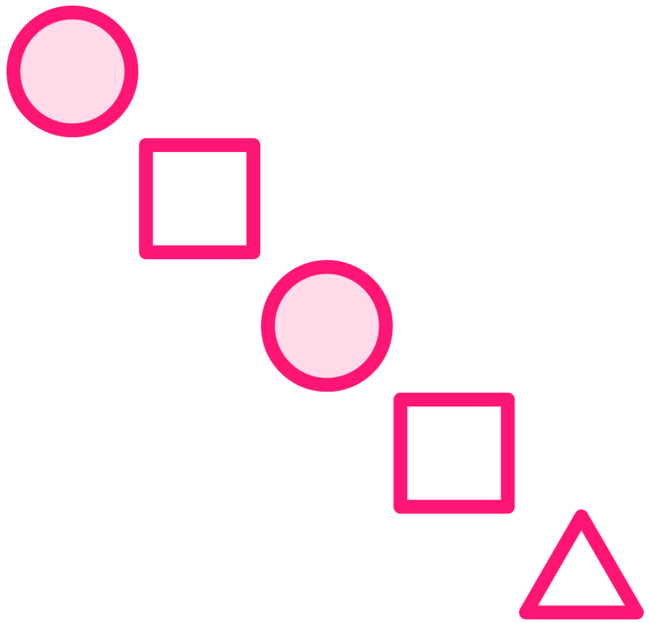
Example:

`{1,2,3}` and `{“one”, “two”, “three”}`

Become

`{ {1,”one”}, {2, “two”}, {3, “three”}}`

Pattern Matching



Building block for many language constructs

Can be used as a simple switch

Or for much more

```
match (n) {
   case 5 => {}
}

match (n) {
   case 5 => {}
   default => {}
}

match (i) {
  case n: i32 => {
    return n;
    }
}
```

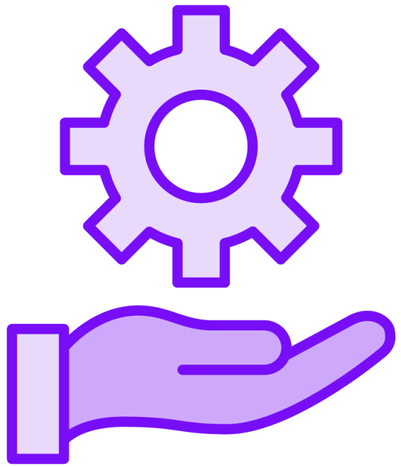
- Note no colon or other punctuation
- You can have a default case
- Not just values for matching, here it’s type

```
fn TypeName[template T:! Type](x: T) ->
String {
  match (x) {
    case _: i32 => { return "int"; }
    case _: bool => { return "bool"; }
    case _: auto* =>
         { return "pointer"; }
    default => { return "unknown"; }
  }
}
```

- This function deduces T and returns a String

- Use _ for a match whose value you don’t need or use
- auto is itself a pattern match

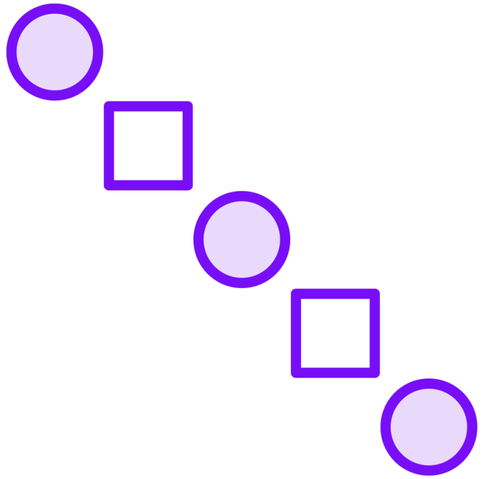
Sum Types (Choice)



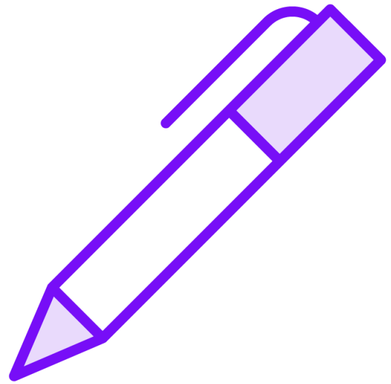
Can hold one of several possible things



Consider std::optional, std::expected, etc.



Pattern matching to inspect what it holds



Can write your own should you need to

```
choice OptionalI32 {
    Some(value: i32),
    None
}

choice Optional(T:! type) {
  Some(value: T),
  None
}

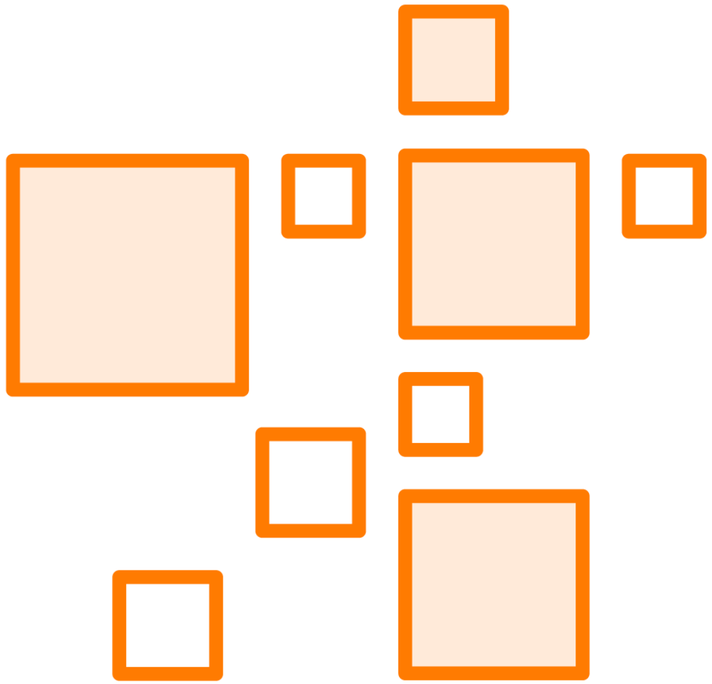
var maybe: Optional(i32) =
    Optional(i32).None;

maybe = Optional(i32).Some(42);

match (maybe) {
  case .Some(i: i32) => {
    Print(i);
  }
  case .None => {
    Print("None");
  }
}
```

- Holds either an i32, or nothing
- Can be generic
- Set it by naming a part if that part has no value
- Or “calling” the part name with a value
- Test with match
- Name for the value is very local

There’s More!



Packages, libraries, and APIs

Lambdas

Operators

- `and`, `or` and `not` (not &&, ||, and !)
- as (conversion)

Dynamic memory allocation (heap)

- Memory safety

, , ,