Language Improvements

Chapel version 1.20

September 19, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- Nilable Class Types
- Class Memory Management
- Task/Forall Intents on 'this'
- Partial Instantiations
- String and Bytes Types
- Private and Public Use
- Use of Top-Level Modules
- Overload Set Checking
- Atomic Interface Stabilization
- Distinguishing 'nothing' from 'void'
- 'class' 'record' and 'enum' as Types
- Changes to 'isSubtype'
- Deprecated Language Features

Nilable Class Types

Nilable Types: Background and This Effort

Background: Nilable types were proposed but not implemented in 1.19

- 'nil' pointers are problematic
  - Tony Hoare calls them "my billion-dollar mistake"
  - 'nil' dereference errors can be difficult to debug
  - other languages are moving towards opting into storing 'nil'
- In 1.19, class instance pointers could be 'nil' and defaulted to 'nil'

```
var x: MyClass;          // stores 'nil'
var y: owned MyClass;    // stores 'nil'
```

This Effort: Implement the proposal and fill in details

Nilable Types: Design

- A class type 'C' means a non-nilable pointer to an instance
  - cannot store 'nil'
  - has no default value
  - including 'borrowed C', 'owned C', 'shared C', 'unmanaged C'
- The type 'C?' is available to opt into being possibly 'nil'
  - 'nil' is the default value
  - including 'borrowed C?', 'owned C?', 'shared C?', 'unmanaged C?'
- 'C' and 'C?' are different types
- The '!' operator unwraps a nilable value, halting if it is 'nil'
- Implicit conversions are allowed from non-nilable to nilable class types

Nilable Types: Example

```
proc getField(x: C) {     // C is a class
  return x.field;         // no check needed here since 'x: C' cannot store nil
}

getField(nil);            // compile-time error: 'getField' expects a non-nilable
var a: C;                 // compile-time error: 'a: C' has no default value
var x: C?;                // ok, use 'nil' as the default value
getField(x);              // compile-time error: x is nilable, passed to non-nilable
getField(x!);             // compiles OK; adds a nil check at runtime
```


The '?' Operator

Nilable Types: the '?' Operator

- The postfix '?' operator on a class type produces the nilable type
  - available only on class types; not on values
- '?' can be combined with a management decorator

```
C;          // non-nilable, generic management
C?;         // nilable, generic management
owned C;    // non-nilable owned
owned C?;   // nilable owned
```

- '?' can combine with 'new' and modifies the type to be created:

```
class C { var x; }
new C(5)? // equivalent to the next line
new C?(5)
```

The '!' Operator

Nilable Types: the '!' Operator

- The '!' is a postfix operator available on class types or values
- It converts the argument to the non-nilable variant
  - and, for 'owned' / 'shared', to the borrowed variant
- When applied to a type, returns the same type as '!' would on a value of that type
- It will halt if it is applied to the 'nil' value (but see 'Open Questions')

If a variable is declared as...	Then x! has type...
`var x: owned C?`	`borrowed C`
`var x: shared C?`	`borrowed C`
`var x: borrowed C?`	`borrowed C`
`var x: unmanaged C?`	`unmanaged C`

Implicit Conversions for Nilability

Nilable Types: Implicit Conversions

- A value of non-nilable type can coerce to an argument expecting nilable

```
var myC: owned C = new C();
proc f(in arg: owned C?) { ... }
f(myC); // coercion and ownership transfer (but see 'Open Questions')
```

  - Note that 'ref' / 'const ref' formals do not allow coercions:

```
proc g(const ref arg: owned C?) { ... }
g(myC); // error: cannot coerce for const ref intent
```

- In contrast, type arguments accept subtypes but do not allow coercions
  - so cannot pass the type 'C' to a 'type t: C?' argument

Nilable Types: Implicit Conversions

- In some cases, implicit conversions combine with generic instantiation

```
var myC: owned C = new C();
proc h(arg: borrowed class?) { ... }
h(myC); // h instantiates with 'borrowed C?'
        // and then 'myC' implicitly converts to 'borrowed C?'
```

- See also the section Generic Types 'class' 'record' and 'enum'

Changes to Casting

Nilable Types: Casts from c_void_ptr

- Casts from 'c_void_ptr' to 'borrowed C' were previously allowed, e.g.:

```
class C { }
var myC = new C();
var cptr: c_void_ptr; // defaults to 'nil'
cptr = myC.borrow() : c_void_ptr;
var c = cptr: borrowed C;
```

- The last line no longer sensible:
  - 'c_void_ptr' can store 'nil' but 'borrowed C' cannot

Nilable Types: Casts from c_void_ptr

- To focus the language design and implementation effort, limited cases supported

- Now 'c_void_ptr' can only be cast to an unmanaged nilable type, e.g.:

```
var c = cptr: unmanaged C?;
```

- Use more casts or '!' to go from there to the appropriate type

```
var c2 = (cptr: unmanaged C?) !;          // halts upon nil
var c3 = (cptr: unmanaged C?) : class;    // throws upon nil, see later
```


Nilable Types: Class Downcast

- Previously, downcast resulted in 'nil' if the value did not have compatible type

```
class Parent { }
class Child : Parent { }
proc f( arg: borrowed Parent ) {
  var asChild = arg: borrowed Child;
  if asChild != nil {
    // do something special for the Child type
  }
}
```

- This no longer makes sense now that 'borrowed Child' cannot store 'nil'

Nilable Types: Class Downcast

- This code can work as before if the target type of the cast is nilable:

```
class Parent { }
class Child : Parent { }
proc f( arg: borrowed Parent ) {
  var asChild = arg: borrowed Child?;
  if asChild != nil {
    // do something special for the Child type
  }
}
```

- Quite a few more cast patterns were added to work with nilable types

Nilable Types: More Class Downcasts

```
var p: borrowed Parent = ...;
p:borrowed Child    // throws if runtime type is not compatible
p:Child             // same as above; infers target management
p:Child?            // results in 'nil' if runtime type is not compatible
                    // (does not throw)

var q: borrowed Parent? = ...;
q:borrowed Child    // throws if runtime type is not compatible or q==nil
q:Child             // same as above; infers target management
q:Child?            // results in 'nil' if q==nil or runtime type is not compatible
                    // (does not throw)
```


Nilable Types: Upcasts and Nilable

```
var c: borrowed Child = ...;
c:borrowed Parent   // class upcast always succeeds (as before)
c:Parent            // same as above; infers target management
c:Child?            // always succeeds; converts to nilable type
c:class?            // always succeeds; converts to nilable type
c:Parent?           // as above but combines with class upcast
var d: borrowed Child? = ...;
d:borrowed Child    // throws if d==nil
d:class             // throws if d==nil; infers target management
d:Child             // same as above; infers target management
d:Parent            // as above but combines with class upcast
```


Nilable Types: Casting Owned and Shared

- Similar casts work with 'owned' and 'shared'
  - upcast
  - downcast
  - cast to change nilability

- For example:

```
var x: owned Child? = ...;
x:class                 // throws if d==nil; infers target management
x:Parent                // as above but combines with class upcast
```

Checking if a Type is Nilable

Nilable Types: Checking if a Type is Nilable

- To support generics with nilable types, added two functions to Types modules:
  - 'isNilableClass' returns 'true' for any class type that can store 'nil'
  - 'isNonNilableClass' returns 'true' for any class type that cannot store 'nil'
  - both work…

    …for 'owned', 'shared', 'unmanaged', and 'borrowed' variants

    …for types and for values
- New support for 'class' as a generic type enables several other patterns:

```
isSubtype(t, class?)        // equivalent to 'isNilableClass(t)'
isSubtype(t, class)         // equivalent to 'isNonNilableClass(t)'
isSubtype(t, owned class)   // returns 'true' if 't' is non-nilable 'owned'
```


Open Questions

Nilable Types: Open Questions

- Does '!' always halt when given 'nil' or can the check be omitted on '--fast' ?
- Should '!' applied to an 'unmanaged' result in the 'borrowed' type?
- Should we support features inspired by the Swift conveniences?
- Should we generalize nilable types to non-class types?
- Should it be possible to initialize a non-nilable variable in a separate statement?

Nilable Types: '!' checking and '--fast'

- Proposal for nilability changes indicated that '!' should check even with '--fast'
- However, the 1.20 compiler leaves this check out with '--fast'
- One of the two strategies needs to be adopted:
  - nil-checks for '!' are left out with '--fast'
    - consider adding another operator that always checks
  - nil-checks for '!' are always included, even with '--fast'
    - need to investigate performance impact and attempt to reduce it
- Strategy under discussion in issue #13603

Nilable Types: '!' and 'unmanaged'

- For 'owned' and 'shared', '!' results in the borrowed type by design
  - a common case with '!' is to call methods, e.g.

```
myOwnedNilable!.method();
```

  - in this situation, ownership transfer would be surprising
- For 'unmanaged' the compiler is inconsistent:

```
type t = unmanaged C?;
var x: t;
x! // unmanaged C
t! // borrowed C
```

- Should both result in 'unmanaged C' or 'borrowed C' ? Discussion in #14092

Nilable Types: Convenience Features

- We expect to add convenience features inspired by Swift

```
if let notNil = possiblyNil {
   // notNil has the non-nilable class type and cannot store 'nil'
}

// if possiblyNil is 'nil', returns 'nil', otherwise computes someMethod()
possiblyNil?.someMethod()

// supplies a default value to use when possiblyNil is 'nil'
possiblyNil ?? default
```

Nilable Types: Conditional Guard

- Should conditional guards introduce a new variable?
  - pro: each variable has a single type

```
if let notNil = possiblyNil {
    // notNil has the non-nilable class type and cannot store 'nil'
}
```

- Or should the compiler just know that the variable is not nil within the conditional?
  - pro: uses existing syntax

```
if possiblyNil {
    // compiler knows that possiblyNil is not 'nil'. Is the type of possiblyNil nilable here?
}
```

- Currently leaning towards introducing a new variable

Nilable Types: Generalizing to Non-Class Types

- Would like to generalize nilable types to a general 'option' type, e.g.
  - an optional 'int', storing an integer value or 'nil':

```
var x: option(int);
var x: int?; // perhaps this would be equivalent to the above
```

  - optional records, storing a record value or 'nil':

```
var y: option(R);
var y: R?;    // perhaps this would be equivalent to the above
```

- Swift offers a precedent for this

Nilable Types: Initializing in Separate Statement

- Currently, a variable of non-nilable type needs to be initialized when declared:

```
var x: owned C;          // error
var x: owned C = f(); // OK
```

- Should the first case be allowed if 'x' is provably not used before it is initialized?

```
var x: owned C;
...; // code not using 'x'
x = new owned C();
```

- This pattern is currently common with arrays, e.g.:

```
var B:[D] owned C;
forall i in D do B[i] = new owned C();
```

- This pattern is also common in initializers to initialize array fields

Incomplete Checking

Nilable Types: Incomplete Checking

- The type checking for nilable types is incomplete in 1.20
- We expect to resolve this in the next release

- Two cases should produce compilation errors but do not yet:
  - arrays containing elements of non-nilable types that are not initialized
  - ownership transfer from a non-nilable owned that may be used again

Nilable Types: Problem with Array Elements

- The compiler currently has incomplete nilable type checking for arrays
- Array elements can be 'nil' when the element type is non-nilable

```
var A:[1..10] borrowed C;  // allowed, even though elements store 'nil'
```

- This is a bug

- The compiler should insist such an array is initialized:

```
var A:[1..10] borrowed C = f();  // OK
```

Nilable Types: Transfer from Non-Nilable Owned

- The compiler currently allows ownership transfer from any 'owned'
  - including non-nilable 'owned'

- Ownership transfer leaves an 'owned' storing 'nil'
  - so this is a hole in the nil-checking

- For example:

```
var x: owned C = new owned C();
var y = x;    // ownership transfer; leaves 'x' nil
x.method();   // dereferences 'nil' — can we catch it at compile time?
```

Nilable Types: Transfer from Non-Nilable Owned

- Here are 3 potential directions for a solution:
  - Do nothing and rely on run-time checks
  - Make ownership transfer from non-nilable a compilation error
  - As above, but error only if the 'owned' might be used afterwards

Nilable Types: Always an Error

- What if ownership transfer from non-nilable 'owned' is a compilation error?
- Some common patterns would be difficult to express cleanly

```
proc makeC() {
  return new owned C();
}
var x = makeC();
// during compilation, it is:
temp t: owned C = makeC();
var x = t; // ownership transfer leaving 't' storing 'nil'
```

- Allowing this case would amount to allowing transfers from temporaries
  - but what is so special about temporaries?

Nilable Types: Always an Error

- Only supporting transfers from temporaries might prohibit simple examples:

```
proc f() {
  var x: owned C = makeC();
  var y = x;  // ownership transfer from x to y
}
```

- Intuitively this is OK since 'x' is non-nil everywhere it is used

- Suggests considering more general rules of when a variable is dead
  - See section in Ongoing Efforts and issue #13704

Nilable Types: Impact, Status, Next Steps

Impact: We can start to move beyond this big language change

Status: Implemented in 1.20; some bugs and open questions remain

- '--legacy-classes' flag available to mostly revert to 1.19 behavior

Next Steps:

- Fix bugs and migrate ~24 tests to the new behavior
- Address the two gaps in checking
- Address open questions

Class Memory Management Improvements

Classes: Background

- Chapel has features to enable automatic memory management of classes
  - 'owned', 'shared', 'borrowed' and 'unmanaged'
- However certain language design questions remained open
- Uncertainty remained around these topics:
  - 'new C()' and 'new borrowed C()' for a 'class C'
  - 'new T()' where 'T' is a type argument
  - behavior of untyped arguments instantiated with 'owned'/'shared'
  - type of 'this' in a type method invoked on 'owned C'

Classes: This Effort

- Address these open questions with the following language adjustments:
  - for a 'class C', 'new C()' results in an 'owned C'
  - undecorated class types have generic management
  - default intent for 'owned' and 'shared' is now 'const ref'
  - removed special rule for untyped arguments instantiated with 'owned'/'shared'
  - stacking management decorators is no longer legal
  - type methods on classes now work for any management and any nilability

'new C()'

Classes: 'new' in 1.19

- In 1.19:
  - 'C' by itself meant 'borrowed C'
  - 'new C()' was equivalent to 'new borrowed C()'
    - which is equivalent to '(new owned C()).borrow()'

- Chose this to make the following statements have the same meaning:

```
var x = new C();
var x: C = new C();
var x: borrowed C = new borrowed C();
```


Classes: Problems with 'new' in 1.19

- 'new C()' being the same as 'new borrowed C()' presented some problems:
  - some programmers found that default and syntax unintuitive and confusing
  - the following common patterns needed 'owned' or 'shared'
    - default of 'new borrowed' caused immediate deletion

```
var x = [i in 1..3] new C(i);

record R { var field = new C(); }

proc f() { return new C(); }

throw new C()
```

Classes: 'new' in 1.20

- Now 'new C()' is equivalent to 'new owned C()':
- This enables all of the following common patterns to work:

```
var x = [i in 1..3] new C(i);

record R { var field = new C(); }

proc f() { return new C(); }

throw new C()
```

- And it reduces the need for 'new borrowed'
  - so we could remove it with less impact on the language

Undecorated 'C'

Classes: Undecorated was 'borrowed' in 1.19

- In 1.19, an undecorated 'C' was equivalent to 'borrowed C':

```
var x: C; // is equivalent to:
var x: borrowed C;
```

- Motivation: 'borrowed' is more common than any other management

- However this strategy led to some problems...

Classes: Problems with undecorated is 'borrowed'

- Default of 'borrowed C' was unsatisfying in the context of field declarations

- When declaring fields, users were more likely to want an 'owned' field

```
record wrapper {
  var field: Implementation;
}
new wrapper(new owned Implementation())
// lifetime error in 1.19 — 'field' is of type 'borrowed Implementation'
```

Classes: Problems with undecorated is 'borrowed'

- Changing 'new C()' to be 'new owned C()' brought about two problems:
  - Inconsistency in type inference:

```
var x     = new C();     // type of x is inferred to be 'owned C'
var x: C = new C();      // type of x 'borrowed C'
```

  - Left the following example unclear and confusing:

```
proc factory(type t) {
  return new t();
}
factory(owned C);        // = 'new owned C()'
factory(C);              // = 'new owned C()' or 'new borrowed C()' ?
factory(borrowed C);     // = 'new owned C()' or 'new borrowed C()' ?
```

Classes: Undecorated is Generic in 1.20

- Now an un-decorated 'C' indicates generic management
  - that is, it is a generic type leaving the management unspecified

- This change addresses all 3 problems

- Field example now works as expected:

```
record wrapper { var field: Implementation; }
new wrapper(new Implementation())
// works as expected in 1.20 — 'field' is of type 'owned Implementation'
```


Classes: Undecorated is Generic in 1.20

- Type inference problems with 'new' are resolved:

```
var x     = new C();          // type of x is inferred to be 'owned C'
var x: C  = new C();          // type of x is inferred to be 'owned C'
var x: C  = new owned C();    // type of x is inferred to be 'owned C'
var y: C  = new shared C();   // type of x is inferred to be 'shared C'
```


Classes: Undecorated is Generic in 1.20

- Factory example now has clear behavior:

```
proc factory(type t) {
  return new t();
}
factory(owned C);       // = 'new owned C()'
factory(C);             // = 'new C()' = 'new owned C()'
factory(borrowed C);    // = 'new borrowed C()'
```

- Now each case behaves the same as textually substituting the type argument

Classes: Behavior not Changed

- Function signature still indicates whether ownership transfer is possible
  - now 'in' intent is the only indicator

- Methods on classes still use 'borrowed' type for 'this'
  - see next slide

Classes: Methods

- Methods on classes still use the borrow type for 'this':

```
class C {
  proc primary()    { /* this.type == borrowed C */ }
}
proc C.secondary() { /* this.type == borrowed C */ }
```

- Changing these to generic management would interfere with overriding
  - requiring significant implementation changes to virtual dispatch
  - creating more combinations of overriding and overloading
- Perhaps a parenthesized type expression should allow generic management:

```
proc (C).secondary() { /* this.type management from call site */ }
```


Passing 'owned' and 'shared' Actual Arguments

Classes: 'owned'/'shared' Actual Arguments in 1.19

- In 1.19, the default intent for 'owned' / 'shared' was 'in'
  - Ownership transfer not surprising since formal argument indicates 'owned'

```
give(new owned C());

proc give(a: owned C) {} // 'a' takes over ownership from the actual arg
```

- 'owned C' passed to 'C' formal argument does not cause ownership transfer
  - Ownership transfer in this case would be surprising
  - Call uses coercion from 'owned C' to 'borrowed C'

```
var mine = new owned C();

borrowing(mine);

proc borrowing(c: C) {}   // borrows because 'C' = 'borrowed C'
```


Classes: 'owned'/'shared' Actual Arguments in 1.20

- The default intent for 'owned' / 'shared' is now 'const ref'
  - avoids the need for a special instantiation rule (see next slide)
- 'owned' typed formal arguments need 'in' intent to transfer ownership:

```
give(new owned C());
proc give(a: owned C) {}      // 'const ref' intent -- no ownership transfer
proc give(in a: owned C) {}   // adding 'in' causes ownership transfer
```

- 'owned C' passed to 'C' argument still does not cause ownership transfer
  - formal argument instantiates as 'owned C' then uses 'const ref' intent:

```
var mine = new owned C();
borrowing(mine);
proc borrowing(c: C) {}       // instantiates to 'owned C' passed by 'const ref'
```


Classes: 'owned'/'shared' to Untyped in 1.19

- Special instantiation rule supported untyped arguments
  - Ownership transfer for such untyped arguments would be surprising
  - So, default-intent untyped formal instantiated to a borrowed type:

```
f(new owned C());
proc f(x) {}          // x was instantiated as a borrow
```

- Declaring a type or using an 'in' intent overrode this behavior

```
g(new owned C());
proc g(y: owned) {}   // y took over ownership from the actual arg
h(new owned C());
proc h(in z) {}       // z took over ownership from the actual arg
```

- Rule was hard to remember and required special handling in generic code

Classes: 'owned'/'shared' to Untyped in 1.20

- To make the language more uniform, we chose to remove the special rule
- Default intent of 'const ref' avoids ownership transfer in the untyped case:

```
f(new owned C());
proc f(x) {}        // x argument instantiated 'const ref x: owned C'
```

- Ownership transfer is still available with the 'in' intent

```
h(new owned C());
proc h(in z) {}     // z takes over ownership from the actual arg
```

- This change resolved many problems with generic code in the standard modules

Combining Management Decorators

Classes: Combining Decorators in 1.19

- In 1.19, management decorators could not be combined in one expression:

```
var x: owned unmanaged C; // error
new borrowed owned C();   // error
```

- However they could be combined in generic code
  - in which case, the outermost decorator would override the others:

```
proc f(type t) { return owned t; }
f(borrowed C); // returns 'owned C'
proc g(type t) { return new owned t(); }
g(borrowed C); // returns 'new owned C()'
```


Classes: Combining Decorators in 1.20

- Now management decorators cannot be combined in any circumstance:

```
var x: owned unmanaged C; // error
new borrowed owned C();   // error
proc f(type t)  { return owned t; }
f(borrowed C); // error: 'owned borrowed C'
proc g(type t)  { return new owned t(); }
g(borrowed C); // error: 'new owned borrowed C()'
```


Classes: Combining Decorators in 1.20

- Casts on types are available to override a management decorator
- Allows one to rewrite the examples that are now errors:

```
proc f(type t) {
  return t:owned;
}
f(borrowed C); // 'owned C'
proc g(type t) {
  type ownedT = t:owned;
  return new ownedT();
}
g(borrowed C); // 'new owned C()'
```

Type Methods

Classes: Type Methods in 1.19

- In 1.19, type methods on class C only worked on 'borrowed C'
- To fix, needed to decide upon the type of 'this' in other cases, e.g.:

```
class C {
  proc type typeMethod() {
    writeln(this:string);
  }
}
var x = new owned C();
x.type.typeMethod();      // should it output 'owned C' or 'borrowed C' ?
```

Classes: Type Methods in 1.20

- Now type methods on classes work on any management and any nilability
- The 'this' type is considered generic and instantiated with the actual type:

```
class C {
  proc type typeMethod() {
    writeln(this:string);
  }
}
var myOwned = new owned C();
myOwned.type.typeMethod();     // outputs 'owned C'
(borrowed C?).typeMethod();    // outputs 'borrowed C?'
C.typeMethod();                // outputs 'C'
```


Classes: Impact, Status, Next Steps

Impact: Language stability in this area has significantly improved

Status: Open questions are addressed and solutions are implemented

Next Steps: Gain experience with current language

- Migrate a straggling test or two to the new behavior
- Resolve the remaining open questions based on experience:
  - Should methods on classes always use 'borrowed' for 'this'?
    - Should there be a way to opt into something more generic?
  - Do we need an optimization to remove indirection for 'const ref' 'owned' ?

Task/Forall Intents on 'this'

Task/Forall Intents: Background

Fields were ignored by task/forall intents

- unintuitive for task/forall constructs in methods on a record
- because 'this' is passed by *default intent* i.e. 'const ref' into task functions

```
forall idx in … {
  this.myArrayField[idx] = 5;   // disallowed: 'this' is a 'const ref' here
  writeln(this);
}
```

Task/Forall Intents: This Effort

Treat fields of 'this' as individual variables

- passed by *default intent* into task functions
- only when 'this' is a record

```
forall idx in … {
  this.myArrayField[idx] = 5;   // ok
  writeln(this);
}
```

now can update array fields

remains a 'const ref' reference to the outer 'this'

Task/Forall Intents: Impact, Status

Impact: More intuitive operation on fields of 'this'

Status: The implementation disallows explicit intents on 'this', on fields of 'this'

- in 'with'-clauses of 'forall', 'coforall', 'cobegin', 'begin'
- to avoid potential ambiguity, e.g.,

```
forall idx in … with (in this) {
  myArrayField[idx] = 5; // treat it as an individual variable with 'ref' intent
                         // or as a field of the shadow variable for 'this' ?
}
```

- we may enable these if a compelling use case arises

Partial Instantiations

Partial Instantiations: Background

- Partial instantiations are generic types with some, but not all, fields instantiated

```
record R {
  type T;
  type U;
  ...
}
type RI = R(int); // 'U' is uninstantiated
```

- Partial instantiations are useful in initialization or in argument types

```
proc foo(r_int : R(int)) { ... } // Any 'R' provided 'T' is 'int'
var x : RI = new R(int, real); // limit type of 'x'
var y = new RI(real); // use 'RI' to avoid typing 'int' repeatedly
```


Partial Instantiations: Background

- Partial Instantiations were not supported in Chapel 1.19
  - Special support existed for argument types but was not widely available
- Many design questions remained open
  - How should users create partial instantiations?
  - How to detect partial instantiations?
  - How to handle fields with default values?
  - How do partial instantiations interact with initializers?

Partial Instantiations: This Effort

- Answered design questions
  - Chose a design with minimal impact on existing programs
- Implemented partial instantiations
  - Users can create partial instantiations
  - Users can programmatically identify partial instantiations
  - Generic types can be passed to and returned from functions
  - Partial instantiations can be used with initializers

Partial Instantiations: Simple Instantiations

- Partial instantiations are instantiated like normal generic types
  - The result is still generic and has a type constructor
  - Type constructor has arguments for uninstantiated fields in declaration order
  - Fields can be instantiated in any order (provided there are no dependencies)

```
record R { type X; type Y; param p : int; }
type A = R(int);   // A's type constructor accepts two args for 'Y' and 'p'
type B = A(p=5);   // B's type constructor accepts one arg for 'Y'
type C = B(real); // C is fully instantiated as 'R(int, real, 5)'

record S { type T; param p : T; } // 'p' depends on 'T'
type Bad = S(p=5); // compile-time error, must instantiate 'T' before 'p'
```

Partial Instantiations: Default Values

- Historically, fields with default values are instantiated in the absence of args
  - E.g., 'range' meant 'range(int, BoundedType.bounded, false)'
- In 1.20 users can pass '?' to indicate that such fields should stay uninstantiated
  - Minimizes impact on existing code
  - Similar to existing feature for formal types

```
record R { type T = int; param S = false; }
proc foo(arg : R); // accepts only 'R(int, false)'
proc foo(arg : R(?)); // accepts any 'R'
proc foo(arg : R(real, ?)); // any 'R" where 'T' is a 'real'
var r : range(?) = 1..10 by 2; // new: initialize ranges of arbitrary types
type RI = R(S=?); // new: 'S' is explicitly uninstantiated, 'T' defaults to 'int'
```

Partial Instantiations: Default Values (continued)

- In 1.20 '?' applies to **all** remaining fields with default values for convenience
  - 'range(?)' vs 'range(?, ?, ?)'
- '?' can only appear once in a type constructor call
  - Cannot be used as a positional argument
  - Must use named expressions after '?'

```
record R { type X = ...; type Y = ...; type Z = ...; }
R(?, ?, int); // compiler-error; '?' used as positional arg
R(uint, real, ?); // OK because '?' is last
R(?, Y=complex); // OK because only named expressions after '?'
R(Y=complex, ?); // Can use named expressions before '?' if desired
```

Partial Instantiations: Querying Fields

- Users have a few options for identifying partial instantiations
- Basic detection is provided by 'isGeneric'

```
record R { type T; param p : int; }
writeln(isGeneric(R(int))); // true
```

- Reflection provides 'isFieldBound'
  - Useful when field names are known

```
use Reflection;
type RI = R(int);
writeln(isFieldBound(RI, "T")); // true
```

Partial Instantiations: Querying Fields

- Alternatively users can compare fields against '?'
  - Here '?' acts as either a type or param expression when needed

```
record R { type T; param p : int; }
type RI = R(int);
writeln(RI.T == ?); // false
writeln(RI.p == ?); // true
```

Partial Instantiations: Passing/Returning Generics

- Generic types can now be passed to and returned from functions
  - For consistency with concrete and fully instantiated types

```
record R { type T; param p : int; }
// Passing and returning fully-generic types
proc getR() type return R;
proc make(type Base, type T, param p : int) return new Base(T, p);
var x = make(getR(), int, 5); // creates an 'R(int,5)'

// Passing and returning partial instantiations
proc RInt type return R(int);
proc make(type Base, param p : int) return new Base(p);
var y = make(RInt, 5); // creates an 'R(int,5)'
```

Partial Instantiations: New-Expressions

- Type aliases can be used with new-expressions
  - Compiler inserts named-expressions for each instantiated field

```
record R { type T; type U; var x : T; var y : U; }
type RIS = R(int, string);
var x = new RIS(5, "hello");
// becomes... new R(T=int, U=string, 5, "hello");
```

- In 1.20 the compiler provides similar support for partial instantiations

```
type RI = R(int);
var y = new RI(string, 5, "hello");
// becomes... new R(T=int, string, 5, "hello");
```

Partial Instantiations: init=

- 'init=' can now support initializing variables with generic type expressions:

```
record Wrapper { type T; var x : T; }
var x : Wrapper(int(8)) = 5; // creates a Wrapper(int(8)) from bound 'T'
var y : Wrapper = 10; // creates Wrapper(int(64)) inferred from '10.type'
```

- To do so, 'init=' uses 'this.type' to compare fields against '?' for the generic case

```
// a user-defined 'init=' method for the Wrapper type
proc Wrapper.init=(x : integral) {
  this.T = if this.type.T != ? then this.type.T else x.type;
  this.x = x:this.T;
}
```

Initializers and Generic Types: Next Steps

Initializers and Generic Types: Next Steps

- Gather feedback on partial instantiations
- Design and implement 'noinit'
  - Allows users to disable initialization of a variable
- Update array and domain implementation to use 'init='
- Allow type aliases to be used with new-expressions involving typeless fields
  - Can possibly leverage new partial instantiation features

String and Bytes Types

Strings and Bytes: Background and This Effort

Background:

- Chapel's string type recently switched to using UTF-8 encoding
  - Currently, POSIX environment variables need to specify a UTF-8 locale
- Several design decisions had also been made but not implemented
  - Index by codepoints by default
  - Create a string-like type to hold arbitrary binary data

This Effort:

- Complete 'string' changes to support Unicode
- Add 'string' factory functions
- Add a new 'bytes' type

String and Bytes: Unicode Support

- Indexing, iteration and length now use codepoint units by default

```
var str = "événement"; // In UTF-8; c3 a9 76 c3 a9 6e 65 6d 65 6e 74
var len = str.length; // len is 9
```

- Use 'string.numBytes' to get number of bytes

```
var len = "événement".numBytes;        // len is 11
```

- 'byteIndex' type is now available to request byte indexing

```
var s1: string = str[3]; // gets the 3rd codepoint as a string: "é"
var s2: string = str[3:byteIndex]; // gets the codepoint starting at the
                                   // 3rd byte as a string: "v"
```


String and Bytes: Factory Functions

- Previously, string initializers used flags to control ownership and copy behavior
  - For example:

```
// Create a string taking ownership of the buffer of myCPtr.
// When the string goes out of scope, the buffer will be freed.
var s = new string(cPtr, length=N, size=N+1, isowned=true,
                   needToCopy=false);
```

  - Error prone: 'isowned=false', 'needToCopy=true' leaks the buffer
- These initializers are now deprecated in favor of the following factory functions:

```
createStringWithNewBuffer(cPtr, length=N, size=N+1);    // copy and dealloc
createStringWithOwnedBuffer(cPtr, length=N, size=N+1);  // no copy but dealloc
createStringWithBorrowedBuffer(cPtr, length=N, size=N+1); // no copy, no dealloc
```

String and Bytes: New 'bytes' type

- The new 'bytes' type stores arbitrary bytes
  - It does not have to be Unicode or character data at all

```
var myBytes = b"some bytes";     // myBytes is a bytes
```

- Has similar methods to string, such as 'find', 'replace', 'join' etc.

- Also supports character-based methods, such as 'toUpper', 'isTitle' etc.
  - These operations assume ASCII encoding and ignore non-ASCII bytes

- Better performance on ASCII text, as there is no need for character decoding

String and Bytes: 'bytes' keyword and literals

- 'bytes' is now a keyword

```
var myBytes: bytes;    // default-initialized bytes
```

- 'bytes' can be created using the following literals

```
var myBytes = b"bytes\nwith quotes";
writeln(myBytes);    //prints: bytes
                                with quotes
```

- Triple quotes can be used to create multiline and/or unescaped 'bytes'

```
var myBytes = b"""raw bytes\nwith triple quotes""";
writeln(myBytes);    //prints: raw bytes\nwith triple quotes
```

- As with strings, single quotes can also be used to specify bytes literals

String and Bytes: I/O with 'bytes'

- 'bytes' type work with I/O calls:

```
var b = b"événement";
writeln(b);    // prints: événement
```

- Bytes also support formatted I/O

```
writef("%ht\n", b);    //prints: b"\xC3\xA9v\xC3\xA9nement"
```

- Binary formats can be used to read/write bytes

```
var b: bytes;
readf("%|4s", b);    // read 4 bytes into b
writef("%|4s", b);  // write 4 bytes from b
```


String and Bytes: Conversions

- A 'string' can be cast to 'bytes', because bytes can hold arbitrary data

```
var myBytes: bytes = "some string":bytes;
```

- To create a string from a bytes, it needs to be decoded

```
var myString: string = b"some bytes".decode();
```

  - Behavior for decoding errors can be specified, e.g.:

```
myBytes.decode(decodePolicy.strict);   // throw DecodeError (default)

myBytes.decode(decodePolicy.replace);  // replace invalid bytes with the
                                       // UTF-8 replacement character

myBytes.decode(decodePolicy.ignore);   // drop invalid bytes
```

- Currently only UTF-8 is supported, but we plan to add more encodings

String and Bytes: Impact

- Better support for UTF-8 strings
- New 'bytes' type holds arbitrary bytes
- Easy-to-use factory functions instead of error-prone initializers

String and Bytes: Open Questions

- How and when are errors with invalid UTF-8 sequences reported?
- How to handle POSIX filenames?
  - Filenames are not necessarily UTF-8 but often will be
- Should the I/O system support conversion between character sets?
  - e.g., to address garbles when printing UTF-8 data to a non-UTF-8 terminal
- Should Chapel source code support non-ascii identifiers? e.g.

```
var événement = 1;
```

- What should different ways to index a 'bytes' return?

```
myBytes[5]; // returns another 'bytes' today, but maybe should return uint(8)?
for b in myBytes do … ;   // yields 'bytes' but maybe should yield 'uint(8)'s?
```


String and Bytes: Next Steps

- Resolve the open questions
- Implement UTF-8 validation for strings
- Fail at program startup if environment is incompatible with UTF-8 encoding
- Retire 'c_string'
- Improve the 'bytes' implementation
  - support param 'bytes'
  - add I/O convenience functions such as 'readbytes'
  - identify and resolve any performance problems

Private and Public Use Statements

Use Statements: Background

- ‘use’ statements are transitive by default
  - Symbols made available to a module are available to its clients, too

```
module Foo { var someVar = 10; }
module Bar { use Foo; }
module Baz {
  use Bar;
  someVar = 1; // Refers to ‘Foo.someVar’ and is currently allowed
}
```

- Possible to hide symbols brought in by ‘use’ in some cases
  - By moving the ‘use’ statement inside function bodies or local scopes …

  … but doesn’t help with symbols needed in arguments/return types

Use Statements: This Effort

- Added support for declaring ‘use’ statements as ‘public’ and ‘private’
  - ‘public use’ means symbols are available to clients of the current module
  - ‘private use’ makes the symbols only visible to the scope of the ‘use’

```
module Bar { private use Foo; }
module Baz {
  use Bar;
  someVar = 1; // Can’t reference ‘someVar’ any more: Bar’s ‘use’ is private!
}
```

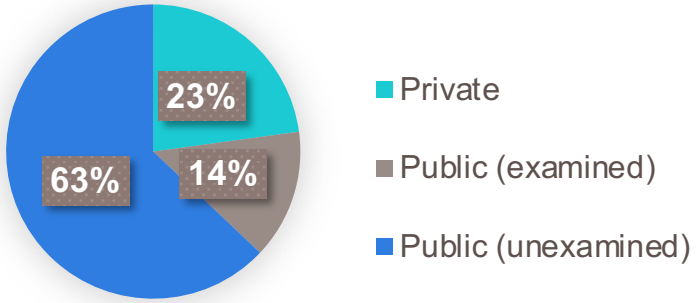
  - ‘use’ without public/private is currently the same as ‘public use’
- Applied ‘private use’ to many internal/standard/package ‘use’ statements

Use Statements: Impact

- Allows libraries to more intentionally control available symbols
  - About 25% of internal module uses no longer leak symbols (see chart)
- Addresses a problem with variables matching library symbols

```
var e = 17;
{
  use Foo;      // Foo has implicit 'use Math'
  writeln(e);   // used to refer to Math.e
}
```

Uses in Chapel Libraries Today



Use Statements: Next Steps

- Continue to convert more public uses to private where reasonable
- Consider changing default ‘use’ to mean ‘private’ for safety
- Add ‘import’ keyword
  - To allow qualified access to a module's symbols without unqualified access

```
import Foo;
writeln(Foo.someVar); // allowed by ‘import’ statement
writeln(someVar);     // not allowed by ‘import’ statement
```

- Resolve open language design questions around ‘use’/‘import’
  - See section in Ongoing Efforts

Use of Top-Level Modules

'use' of Modules: Background

- Traditionally, if a top-level module was known to 'chpl', it could be named in code

```
module M {
  N.foo();   // referring to N is OK since it's visible through lexical scoping
}

module N {
  proc foo() { … }
}
```


'use' of Modules: Background

- However, this also resulted in some surprising / inconsistent behaviors…

```
module M {
  N.foo();   // error: compiler has no notion of what `N` is
}
```


'use' of Modules: Background

- However, this also resulted in some surprising / inconsistent behaviors…

```
module M {
  N.foo();   // whether or not `N` makes sense here depends on O's contents…
  {  use O;  }
}
```

'use' of Modules: Background

- However, this also resulted in some surprising / inconsistent behaviors…

```
module M {
  N.foo();   // error: compiler still has no notion of what `N` is
  {  use O; }
}

module O {
  proc bar() { … }
}
```


'use' of Modules: Background

- However, this also resulted in some surprising / inconsistent behaviors…

```
module M {
  N.foo();   // OK: O's use of `N` adds it as a top-level module, so now it's visible
  {  use O; }
}

module O {
  use N;
  proc bar() { … }
}
```

N.chpl

```
module N {
  proc foo() { … }
}
```

'use' of Modules: Background

- However, this also resulted in some surprising / inconsistent behaviors…

```
module M {
  N.foo();   // Still OK: O's use of `N` may be private, but N is still lexically visible
  {  use O; }
}

module O {
  private use N;
  proc bar() { … }
}
```

N.chpl

```
module N {
  proc foo() { … }
}
```

'use' of Modules: This Effort

- This release adopts a new rule to avoid this instability:
  - To refer to a top-level module, a 'use' of that module must be lexically visible

```
module M {

  N.foo();   // this reference to N is no longer OK since there is no `use` of it
}

module N {
  proc foo() { … }
}
```

'use' of Modules: This Effort

- This release adopts a new rule to avoid this instability:
  - To refer to a top-level module, a 'use' of that module must be lexically visible

```
module M {
  use N;
  N.foo();    // now it's OK due to the lexically visible `use N`
}

module N {
  proc foo() { … }
}
```

'use' of Modules: Impact, Next Steps

Impact:

- Code that relies on the historical rules needs updating to add 'use's
- Code updated in this way has typically ended up being easier to understand

Next Steps:

- Should the same rule apply to submodules?
  - Ones defined in the current module?
  - Ones defined in a sibling or cousin module?
- Continue efforts to improve modules in their role as namespaces

Overload Set Checking

Overload Sets: Background

Hijacking: when a change to a library affects user code inadvertently

- with a reasonable addition of a function

```
module LibA { // v1.0
  proc draw() {…}


}
module LibB {
  proc drill(x: real) {…}
}
use LibA, LibB;
draw(); drill(1);
```

use LibA to `draw`

initially, user code invokes `LibB.drill` as desired

Overload Sets: Background

Hijacking: when a change to a library affects user code inadvertently

- with a reasonable addition of a function

```
module LibA { // v2.0
  proc draw() {…}
  proc drill(x: int) {…}
}
module LibB {
  proc drill(x: real) {…}
}
use LibA, LibB;
draw(); drill(1);
```

LibA 2.0 adds new functionality

After update to LibA 2.0, this switches to `LibA.drill` without user knowledge

Overload Sets: This Effort

Report an error upon a possibility of hijacking

- upon a function call, all applicable functions must be in the same *overload set*
- an *overload set* contains like-named functions defined in the same module

```
module LibA {
  proc drill(x:int) {…}
  proc drill(x:imag) {…}
}
```

```
module LibB {
  proc drill(x:real) {…}
  proc drill(x:complex) {…}
}
```

ok: only `drill` in LibB is applicable

error: functions in both overload sets are applicalbe

```
use LibA, LibB;  drill(myComplex);  drill(myInt);
```

Overload Sets: This Effort

Overload Sets check passes when one of the following applies:

- all applicable functions are in one overload set
- the best applicable function is in the overload set that is *more visible* than others
- for methods on classes:
  - the best function is in the same overload set as the class declaration

```
use Barriers, AllLocalesBarriers; // AllLocalesBarrier is child of Barrier
(new AllLocalesBarrier()).lock(); // would be an error without this rule
```

Compiler option '--no-overload-sets-check' turns off this check

Further details available in the online technote Checking Overload Sets

Overload Sets: Impact, Next Steps

Impact: out of >10k Chapel tests:

- potential issues were exposed in 3 tests
- multiple overload sets are intentional in 2 tests

Next Steps:

- seek user feedback: what needs adjusting?
- language support for merging overload sets?

```
use LibA, LibB;

merge-overload-sets drill;  // syntax t.b.d.

drill(4.7932);   // use LibB.drill(real)

drill(5);        // use LibA.drill(int); ok that LibB.drill is also applicable
```


Atomic Interface Stabilization

Atomics: Background and This Effort

Background: Chapel atomics were modeled after C11/C++11 atomics

- We identified several interface problems during language stabilization efforts
  - C names were used for memory orders instead of a Chapel enum
  - peek/poke operations were not ready for stabilization
  - compareExchange interface did not match C/C++

This Effort: Addressed the above problems to stabilize the atomics API

Atomics: Memory Order

Background: Previously, we used the C names for atomic memory orders

- e.g. memory_order_relaxed

This Effort: Changed memory orders to a Chapel enum

- Updated atomic operations to require param orders
  - Order must be known at compile-time to optimize; params enforce this

```
enum memoryOrder { relaxed, acquire, release, acqRel, seqCst }

proc AtomicT.add(value: T, param order:memoryOrder = memoryOrder.seqCst) ...

a.add(1);
a.add(1, memoryOrder.relaxed);
a.add(1, memoryOrder.seqCst);
```


Atomics: peek and poke

Background: peek/poke provide non-atomic read/write operations

- These can mitigate overhead when atomicity is not required
  - e.g peek/poke are 200x faster than read/write for ugni network atomics
- However, peek/poke need interface work

This Effort: Moved peek/poke to an unstable opt-in module

- Allows us to improve their API without breaking core atomic interface

```
use PeekPoke;
var T: [1..n] atomic int;
[i in 1..n] T[i].poke(i);
const sum = + reduce T.peek();
```

Atomics: compareExchange: Background

- compareExchange API did not match C/C++

  Chapel:

```
proc compareExchangeStrong(expected: T, desired: T): bool
```

  C++:

```
bool compare_exchange_strong(T& expected, T desired);
```

- In C/C++, 'expected' is passed by reference and updated on failure
  - In Chapel, it is passed by value, so not modified on failure
  - This presented a problem because symmetry with C/C++ is important

Atomics: compareExchange: This Effort

- Decided to adjust the API to match C/C++
  - Renamed compareExchange to compareAndSwap this release
  - Will add compareExchange that matches C/C++ next release
- compareAndSwap simplifies common patterns

```
while !l.compareAndSwap(false, true) { } // exchange would need tmp and assignment in loop
l.write(false);
```

- compareExchange is more efficient when 'expected' has to be updated

```
proc AtomicT.mult(v: T) {
  var oldV = this.read();
  while !this.compareExchange(oldV, oldV * v) { } // swap would need atomic load in loop
}
```


Atomics: Status and Next Steps

Status: Atomic interface has been improved

- Not expecting any further breaking changes

Next Steps: Continue to improve atomic capabilities

- Add a general mechanism to perform non-atomic updates on atomics
  - Enables optimization of HPC apps with distinct init, write, and read phases
  - May look to Rust ‘get_mut’ or C++ ‘atomic_ref’ for inspiration
- Add optimization hints for local vs. remote access
- Allow users to specify required operations (for networks with limited atomics)
- Extend atomic support to classes and records
- Implement operator overloads

Distinguishing 'nothing' from 'void'

Void/Nothing: Background

Background: Two types of 'void'

- A routine that doesn't return anything has type 'void'
  - Result cannot be assigned to a variable

```
proc f(): void { writeln("no return"); }
var a = f(); // error: illegal use of a function that doesn't return a value
```

- Giving a variable the type 'void' made the compiler remove it

```
var filename: if useFilename then string else void;
```

  - There was also a value of type 'void' that could be returned and assigned

```
proc g(): void { return _void; }
var filename = if useFilename then "myfile.txt" else g();
```


Void/Nothing: This Effort, Impact

This Effort: Break the two concepts apart into 'void' and 'nothing'

- The 'nothing' type has a single value: 'none'

```
proc f(): void { writeln("no return"); }
proc g(): nothing { return none; }
var fn: if useFilename then string else g();
```

Impact: Naturally separates two distinct concepts

- Removes ambiguity around routine returns
- Corresponds roughly to notions of “bottom” vs. “unit” type

Generic Types 'class' 'record' and 'enum'

Generic Types: Background, This Effort

Background: Generic types are sometimes useful for categories of types

- Provide a useful alternative to 'where' clauses
  - faster to compile in practice
- 'enumerated', 'integral', and 'numeric' have existed for this purpose

This Effort: Added 'class', 'record', and 'enum' generic types

```
proc a( x: class ) { }   // x accepts any non-nilable class
proc b( y: record ) { }  // y accepts any record
proc c( z: enum ) { }    // z accepts any enum
```


Generic Types: Impact, Next Steps

Impact: Easy and intuitive to express particular management or nilability

```
proc f( x: class ) { }              // any non-nilable class
proc g( x: class? ) { }             // any nilable class
proc h( x: owned)   { }             // nilable or non-nilable owned
proc i( x: borrowed class ) { }     // any non-nilable borrowed class
proc j( x: shared class? )  { }     // any nilable shared class
```

Next Steps:

- Deprecate 'enumerated' in favor of 'enum'
- Consider adding user-defined type classes

Changes to 'isSubtype'

Subtypes: Background

- 'isSubtype' has included coercions since 1.18

- However class inheritance is really different from numeric conversion:
  - passing an 'int(8)' to an 'int' formal argument creates a new value
  - passing a 'Child' class to a 'Parent' formal argument aliases the original

- Would like to support type queries to identify these cases

Subtypes: This Effort

- Split 'isSubtype' into 'isSubtype' and 'isCoercible'
  - 'isSubtype' no longer considers coercions
  - '<=' on types is still equivalent to 'isSubtype'
    - so, it no longer considers coercions
- The term 'subtype' now has a more concrete meaning:
  - an implicit conversion to a subtype refers to the original memory
  - vs. a coercion which produces a new value of the target type
- In principle, subtypes can be passed to 'const ref' arguments
  - but this is not implementable until we move to '--llvm' by default

Subtypes: Impact, Next Steps

Impact: Several language design elements are on firmer ground

- 'type' arguments allow subtyping but not coercion
- applied subtype definition to new management and nilability conversions
  - conversion to borrowed type or to a parent class type are subtyping
  - conversion from non-nilable to nilable is a coercion

Next Steps:

- Improve language documentation about the meaning of the term 'subtype'
- Consider enabling subtyping conversions for 'const ref' arguments with '--llvm'

Deprecated Language Features

Deprecated Language Features: Background

- To prepare for the Chapel 2.0 release, we're also removing problematic features
- Some notable cases in Chapel 1.20 were:
  - copy-initializers implemented with 'init'
  - string + value operations
  - assignment from ranges to n-dimensional arrays
  - opaque domains and arrays
- See "Deprecated / Removed Language Features" in CHANGES.md for others

Deprecating Copy Initializers

Deprecating Copy Init: Background, This Effort

Background: 1.19 introduced the 'init=' method for copy initialization

- Allowed users to implement either 'init' or 'init=' methods for copy initialization
  - Kept existing code working
  - Allowed users to explore 'init='

This Effort: Added deprecation warning if 'init' is resolved as the copy initializer

```
9  proc R.init(other:R) {
10   this.x = other.x;
11 }
```

```
foo.chpl:9: warning: 'init' has been deprecated as the copy-initializer, use 'init=' instead.
```


Deprecating Copy Init: Impact, Next Steps

Impact: Most programs simply require changing text from 'init' to 'init='

- Some programs used 'init' for new-expressions and copy-init
  - Needed to write a new 'init=' method

Next Step: Change the warning to an error

String + Value Operations

String + Value: Background, This Effort

Background: Chapel has traditionally supported *string* + *value* operations

- these were supported for basic scalar types: bools, ints, reals, etc.

```
[i in 1..10] writeln("hello " + i);
```

This Effort: Deprecated these '+' overloads due to possible confusion

- the '+' operator is typically only supported on identical types

```
writeln(1 + "10");          // should this be “110” or 11?
```

- since not all types had '+' overloads, led to confusion

```
writeln("hello " + 1..10); // “hello 1..10” or an array of “hello i” strings?
```

- workarounds don’t seem onerous (using casts or creating new '+' overloads)

```
[i in 1..10] writeln("hello " + i:string);
```

String + Value: Impact, Status, Next Steps

Impact:

- existing code must be rewritten to use explicit casts or user overloads
  - all existing tests and released modules have been updated

Next Steps:

- consider other non-orthogonalities and points of confusion in the language

Assignment from Ranges to n-dimensional Arrays

(for n > 1)

Array = Range: Background, This Effort, Status

Background: Chapel has long supported assignments from ranges to nD arrays

```
var A: [1..10, 1..10] int;
A = 1..100;
```

- however, this doesn't match our traditional notion of *matching shape*

This Effort: Deprecated these features now to avoid future surprises

Next Steps: Add utility iterators that can support such patterns

- for example:

```
for (a, i) in zip(flatten(A), 1..100) do a = i;
flatten(A) = 1..100;
```

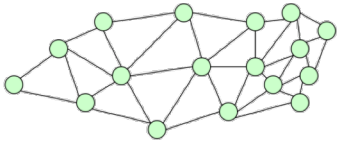

Opaque Domains and Arrays

Opaque Domains: Background, This Effort, Status

Background: Chapel has supported opaque domains / arrays since the outset

- intended to support unstructured collections as domains / arrays

```
var OD: domain(opaque);
var A: [OD] real;
```



- however, in practice, users have always relied upon different approaches
  - e.g., sparse or associative domains / arrays; class-based collections

This Effort: Deprecated these features to reduce code maintenance

- could always reintroduce in the future, if considered valuable

Status: Deprecation warnings added in 1.20; features will be removed in 1.21

For More Information

For a more complete list of language-related changes in the 1.20 release, refer to the following sections of the CHANGES.md file:

- Syntactic/Naming Changes
- Semantic Changes / Changes to the Chapel Language
- New Features
- Feature Improvements
- Deprecated / Removed Language Features
- Standard Library Modules
- Error Messages / Semantic Checks

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?


chapel_info@cray.com


@ChapelLanguage


chapel-lang.org

cray.com


@cray_inc


linkedin.com/company/cray-inc-/