OUTLINE

• Core Language Changes
• Type-related Changes
CORE LANGUAGE CHANGES

• Clarifying Generic Types
• Improvements to Intents
• Special Methods
• Identifier-related Changes
• Generic Numeric Arguments
• Lifetimes of Temporaries
• L-value Improvements
CLARIFYING USES OF GENERIC TYPES
Chapel’s record and class types can be either concrete or generic:

```chapel
record R { // a concrete record: nothing about its type is left unspecified
    var x: int;
}

record RG { // a generic record—the type of ‘t’ must be specified to make it complete
    type t;
    var x: t;
}

record RD { // a generic, but fully-defaulted record—it is complete without more information, yet ‘t’s default could be overridden
    type t = int;
    var x: t;
}
```

Type expressions can also be considered ‘complete’ vs. ‘incomplete’

- **complete**: no additional information is required to create a variable of the type (e.g., ‘R’ and ‘RD( )’)
- **incomplete**: some information is lacking (e.g., ‘RG’ since ‘RG.t’ is not specified)
**GENERIC TYPE CLARITY**

More Background

- Chapel has supported the ability to indicate an incomplete type expression using ‘(?)’

  ```plaintext
  record R { type t;  param p: int; }
  var Rint = R(t=int, ?);  // binds 't', but leaves 'p' unspecified
  proc foo(r: R(?)) { ... }  // indicates that 'fooO' accepts arguments of any flavor of 'R'
  ```

- However, it has also been fairly lax in distinguishing between generic and concrete type expressions
  - Potentially rationalizable as “productive!”, but in practice it can also lead to more confusion than benefit

    ```plaintext
    var a: T1,  // is 'T1' concrete? generic? generic, but fully specified? Do I need to say more in order to make it complete?
    b: T2();  // what about 'T2' here?
    ```

- Lately, we’ve been wrestling with questions like:
  - How much code needs to be inspected to determine whether a given type or procedure is generic?
  - What can we do to reduce this amount of code and improve readability?
This Effort

- Began exploring several rule changes to better distinguish between...
  - concrete and generic types
  - complete and incomplete types

- To gain experience with these changes, we implemented them as warnings for now
  - If popular with users, an upcoming release would make them into errors
  - Note that these warnings are on by default, not specific to ‘--warn-unstable’

- Generally speaking, the new rules tend to involve...
  - warning when incomplete types are written without ‘?’ in their type signatures
  - treating concrete types less like generic types
  - treating complete generic types similarly to concrete types
The following slides walk through a series of scenarios related to generic types:
- Each summarizes the topic, showing what has changed vs. not.
- Most examples are illustrated with the following record types:
  - However, the scenarios are handled similarly for class types.

```plaintext
record R {  // a concrete record: nothing about its type is left unspecified
  var x: int;
}

record RG {  // a generic record—the type of ‘t’ must be specified to make it complete
  type t;
  var x: t;
}

record RD {  // a generic, but fully-defaulted record—it is complete without more information, yet ‘t’s default could be overridden
  type t = int;
  var x: t;
}
```

**GENERIC TYPE CLARITY**

Guide to the following slides
CLARIFYING USES OF GENERIC TYPES

- T(...) Implies Generic Types
- ‘?’ Implies Incomplete Types
- Generic Fields & Stabilization
- Wrap-up
MAKING ‘T(...)’ IMPLY ‘T’ IS GENERIC (HAS A TYPE CONSTRUCTOR)
**GENERIC TYPE CLARITY**

Attempting to call a type constructor for a concrete type

- **Given:**

  ```chapel
  record RG { record RD {
    record R {
      type t;
      var x: int;
    }
  }
  }
  ```

  ```chapel
  record RD {
    record R {
      type t = int;
      var x: t;
    }
  }
  ```

- **In Chapel 1.30 and earlier, a concrete type signature could be specified with parentheses:**

  ```chapel
  var r: R();
  ```

  - **Before:** permitted, but misleading since ‘R’ does not define a type constructor to call
  - **Now:**

    ```chapel
    code.chpl:5: warning: unnecessary type construction call
    code.chpl:1: note: 'R' has no generic fields
    code.chpl:5: note: remove the '()' 
    ```

    - To address such warnings in 1.32, rewrite as:

      ```chapel
      var r: R;
      ```
**GENERIC TYPE CLARITY**
Closed an asymmetry for concrete formal types

- **Given:**
  ```
  record RG { record RD {
    record R { type t; type t = int; var x: int; var x: t; var x: t; }
  }
  }
  ```

- Chapel 1.30 and earlier permitted concrete formals to be specified with ‘(?)’
  ```
  proc foo(r: R(?)) { ... }
  ```

  - **Before:** accepted as though the argument was simply ‘r: R’
  - **Now:**
    ```
    code.chpl:5: error: the formal argument 'r' is marked generic with (?)
    but the type 'R' is not generic
    ```

- To address such warnings in 1.32, rewrite as:
  ```
  proc foo(r: R) { ... }
  ```
MAKING INCOMPLETE TYPES MORE VISIBLE USING ‘?’
**GENERIC TYPE CLARITY**

Specifying incomplete type signatures without arguments

- **Given:**

```
record RG {
    record RD {
        record R {
            type t;
            var x: int;
            var x: t;
        }
    }
}
```

- **In Chapel 1.30 and earlier, generic type constraints could be specified without argument lists:**

```
var r1: RG,
    r2: RG();
```

**Before:** permitted, but misleading since neither ‘RG’ nor ‘RG( )’ is a complete type

**Now:**

```
var r1: RG(?), r2: RG(?);
```

- **To address such warnings in 1.32, rewrite as:**

```
var r1: RG(?), r2: RG(?);
```
**GENERIC TYPE CLARITY**
Specifying fully-defaulted generic type signatures (no change)

- Given:

```chapel
record RG {
    record RD {
        record R {
            type t;
            var x: int;
        }
        var x: t;
    }
    var x: t;
}
```

- As in Chapel 1.30, fully-defaulted generic types can still be declared with or without arguments:

```chapel
var r1: RD,       // OK: a fully-defaulted generic can act like a concrete type
r2: RD(),        // OK: 'RD' defines a type constructor with defaulted arguments
r3: RD(t=real);  // OK: we can override any defaulted arguments
```

- This behavior and the change on the previous slide address an asymmetry in Chapel 1.30 and earlier:

```chapel
var rd: RD ≈ var rd: RD()   // “the fully-defaulted/constrained ‘RD’”
var rg: RG ≈ var rg: RG(?)   // “any ‘RG’”
```

- Now, when we see ‘var x: T;’, we know that ‘T’ is a complete type, either concrete or fully-defaulted
Applications to other contexts

• Given:

  record RG {
    class C {
      type t;
      var x: t;
    }
  }

• Though the previous examples focused on variable declarations, similar warnings apply to:

  • field declarations then:
    record R { var r: RG; }
  vs. now:
    record R { var r: RG(?); }

  • formal argument declarations then:
    proc foo(r: RG) { ... }
  vs. now:
    proc foo(r: RG(?)) { ... }

  • parent class declarations then:
    class D: C { ... }
  vs. now:
    class D: C(?) { ... }
**GENERIC TYPE CLARITY**
Class fields with generic management type

- Given: ```
  class C { … }
  
  record R {
    var c: C;  // this field makes this record generic
  }
```

- Record ‘R’ is generic because no memory management style is specified for its class field, ‘c’
  - Yet, at a glance, it appears to be non-generic

- **Before:** users were often confused by error messages that tended to ensue from such patterns

- **Now:**

  ```
  code.chpl:4: warning: field is declared with generic memory management
  note: consider adding 'owned', 'shared', or 'borrowed'
  note: if generic memory management is desired, use a 'type' field
    to store the class type
  ```

  - To address such warnings in 1.32, introduce a memory management type (or use an explicit type field):
    ```
    var c: owned C;
    ```
• Given generic record ‘RG’, the following cases are unchanged and do not generate warnings today:
  • incomplete type expressions as actual arguments:
    ```
    foo(RG); // OK to pass in an incomplete type without additional notation
    proc foo(type t) { ... }
    ```
  • incomplete type expressions used to define type aliases:
    ```
    type t = RG; // OK to create an alias to ‘RG’ without additional notation
    ```
  • generic method receiver types:
    ```
    proc RG.foo() { ... } // OK to create a new secondary/tertiary method on ‘RG’ without additional notation
    ```
  • formals or variables of class type with generic memory management:
    ```
    var myC: C;
    proc bar(c: C) { ... }
    ```
AVOIDING GENERIC FIELD CASES
NOT READY FOR STABILIZATION
**GENERIC TYPE CLARITY**

Specifying the types of generic var/const fields

- **Given:**
  ```chapel
  record RX { // or: record RX {
    var x;
    var x: integral;
  }
  }
  ```

- Chapel 1.30 and earlier permitted type signatures to specify generic field types by name
  ```chapel
  var r: RX(x=int) = new RX(x=42);
  ```

  - **Before:** accepted (despite ‘x’ not being a ‘type’ field) and specified the type for field ‘x’
  - **Now:**
    ```chapel
    code.chpl:2: note: field 'x' declared here is not 'type' or 'param'
    code.chpl:5: error: named arguments can only be used in type construction
to set 'type' or 'param' fields
    ```

  - To address such warnings in 1.32, rely on positional arguments instead of named:
    ```chapel
    var r: RX(int) = new RX(x=42);    // OK — 'int' specifies the type of generic field 'x', positionally
    ```
Default initializing generic var/const fields

- **Given:**
  ```chapel
  record RX { // or: record RX {
      var x;
      var x: integral;
  }
  }
  ```

- Chapel 1.30 and earlier permitted default initialization of records with generic fields
  ```chapel
  var r: RX(int);
  ```

  - **Before:** field ‘x’ would take on the default value of the type specified in the constructor call
  - **Now:**
    ```chapel
    code.chpl:5: error: default initialization with type 'RX(int(64))' is not yet supported
    code.chpl:2: note: field 'x' is a generic value
    code.chpl:2: note: consider separately declaring a type field for it or using a 'new' call
    ```

  - Rationale: addresses a pre-existing bug/inconsistency that limited our ability to improve the situation, had it been retained
  - see the following slides for additional details
For a generic record with an explicit type field, default initialization works as follows:

- Given the record type:
  ```
  record RG {
    type t;
    var x: t;
  }
  ```

- A declaration relying on default initialization, like this:
  ```
  var x: RG(int);
  ```

- Is translated into the following call, which relies on the presence of a type argument named ‘t’:
  ```
  var x: RG(int) = RG.init(t=int);
  ```

- This call is compatible with either compiler-generated or user-defined initializers:
  - The compiler-generated initializer looks something like this:
    ```
    proc RG.init(type t, in x: t = defaultOf(t)) { ... }
    ```
  - And to support default initialization, a user can write an initializer like this:
    ```
    proc RG.init(type t) { ... }
    ```

SIDEBAR: DISALLOWING DEFAULT INIT OF GENERIC VAR/CONST FIELDS
• However, when a record has a generic var/const field, challenges arise:
  • Given the record:
    ```
    record RX { var x; }
    ```
  • It’s not clear what the compiler should generate to implement default initialization:
    ```
    var x: RX(int);
    ```
  • One challenge relates to the lack of a named type field to use as an argument name
    – e.g., we could translate into:
      ```
      var x: RX(int) = RX.init(x=int);
      ```
    – but this would be odd since ‘x’ represents a value, not a type
  • Meanwhile, prior to 1.32, the compiler would pass a default value in directly to the compiler-generated initializer:
    ```
    var x: RX(int) = RX.init(int, x=defaultOf(int));
    ```
    – but this is problematic since...
      ...the compiler shouldn’t be determining values of such fields, the initializer itself should be
      ...the user didn’t have a means of writing a similar initializer themselves
  • Rather than preserve this asymmetric / suspect behavior, we chose not to support it for now

SIDEBAR: DISALLOWING DEFAULT INIT OF GENERIC VAR/CONSTRUCT FIELDS
GENERIC TYPES: WRAP-UP
GENERIC TYPE CLARITY

Impact

- Several syntactic asymmetries and points of potential confusion now generate warnings
- After resolving warnings, developers have generally considered the code to be much clearer
  - Overall, the effort to update code has felt worthwhile in terms of the benefits
**GENERIC TYPE CLARITY**

Next Steps

- Get feedback on these warnings from users
  - If reaction is positive, convert warnings to errors
  - If not, devise an alternate plan or revert to the previous behavior
- Consider adding a way to syntactically indicate “generic management type”
  ```
  var c: ? C;
  ```
- Continue to improve support for type constructors:
  - ability for ‘chpldoc’ to document them for compiler-generated cases
  - ability for users to define their own type constructors
  - maintain ability for Dyno to resolve type constructors
- Potentially consider tightening up rules about passing generic types to ‘type’ arguments
  - note that this would be a breaking change
  ```
  proc foo(type t) { var x: t; }
  foo(integral); foo(RG); // permitted today, but causes errors at declaration point of ‘x’ since ‘t’ is generic
  ```
IMPROVEMENTS TO INTENTS
IMPROVEMENTS TO INTENTS

Background

- Chapel uses *ints* in a few places:

  - Formal argument intents:
    ```
    proc foo(const in arg: int) { }  // 'const in' is an argument intent
    proc ref C.bar() { }           // 'ref' is the intent of method 'barO's receiver ('this')
    ```

  - Return intents and yield intents:
    ```
    var x: int;
    proc bar() ref { return x; }    // 'ref' indicates a reference to 'x' is returned rather than its value
    iter baz() ref { yield x; }     // 'ref' indicates a reference to 'x' is yielded rather than its value
    ```

  - Task Intents:
    ```
    forall i in 1..10 with (in z) { }  // task intent indicates each task implementing the 'forall' gets its own copy of 'z'
    begin with (var z=20) { }          // task intents can also create a task-local variable
    ```
IMPROVEMENTS TO INTENTS
This Effort

• Took on a few intent-related efforts:
  • Began changing default intents for arrays and record receivers to ‘const’
  • Updated the definition of ‘const’ intents
  • Introduced an explicit ‘out’ return/yield intent
DEFAULT INTENT CHANGES FOR ARRAYS AND RECORD RECEIVERS
DEFAULT INTENT CHANGES FOR ARRAYS

Background

- Traditionally, the default intent for arrays has been ‘ref if modified’
  - If the array was modified within the scope’s body, the intent was ‘ref’, otherwise ‘const [ref]’

```plaintext
proc myFunc(/*ref*/ A:[], /*const*/ B:[],
             ref C:[], const D[:]) {
    // legal; A is inferred to be 'ref' and 'B' as 'const'
    A = B;
    // legal; C is explicitly 'ref' and 'D' is 'const'
    C = D;
    // illegal; D is explicitly 'const'
    // D = C;
}
```

- Rationale: Believed that modifying arrays fit the principle of least surprise / was part of their nature
  - Yet simply using a default intent of ‘ref’ prevented ‘const’ arrays from being passed to read-only arguments

- The same default intent rules were used for function arguments, task intents, and ‘forall’ intents
DEFAULT INTENT CHANGES FOR ARRAYS
This Effort

• Decided to change the default intent for arguments and task intents to be ‘const’
  • Makes them consistent with most other types
  • Reduces compiler complexity
  • Makes function signatures and parallel operations clearer in terms of what is expected to be modified

```plaintext
var myInt: int;
var myArray: [1..10] int;

coforall i in 2..9 {
  myInt += i;  // error due to attempting to modify a ‘const’ shadow variable
  myArray[i] += 1;  // was: OK because arrays defaulted to ‘ref’; now: similar error due to default of ‘const’
}
```

• However, maintained the exception for ‘forall’ loops (for now) due to concern from users and developers
  • See discussion on subsequent slides for more details
DEFAULT INTENT CHANGES FOR ARRAYS

Impact

• Deprecated modifying a promoted array indexing operation
  • Draws more attention to potential race conditions
  • Can still be written explicitly

```plaintext
const B = [2, 4, 4, 7];
var A: [1..10] int;

// A[B] += 1;                        // was: an unsafe race that would've been permitted; now: an error due to 'A' being a 'const' shadow variable
[b in B with (+ reduce A)] A[b] += 1; // user must now write a loop with explicit intents, ideally resolving the race

proc myFunc(ref c: int) do
  c = 2;

// myFunc(A[B]);                // was: an innocuous race that would've been permitted; now: an error due to 'A' being a 'const' shadow variable
[b in B with (ref A)] myFunc(A[b]); // user can still express the innocuous race using a loop
```
DEFAULT INTENT CHANGES FOR RECORD RECEIVERS

Background and This Effort

Background: The default intent for the record receiver ‘this’ was also ‘ref if modified’

• Rationale: similar to the array case, it seemed natural that methods would modify their fields by default
  – Yet, a default of ‘ref’ prevents read-only methods from being called on ‘const’ records

This Effort: Changed the default intent for record receivers to be ‘const’

• Rationale:
  – makes ‘this’ have the same default intent as any other argument
  – removes any remaining need for ‘ref if modified’ reasoning within the compiler

```plaintext
proc /*const*/ myRecord.get() do return this.x;
proc /*ref*/ myRecord.set(newVal) do this.x = newVal;
```
SECTION 5.3 DEFAULT INTENT CHANGES

Status: forall-loops

- As mentioned earlier, decided to preserve the old ‘ref if modified’ behavior for ‘forall’ loops, for now
  - No warnings by default, but generates similar warnings as other cases with ‘--warn-unstable’

- Rationale: Concerns from users and developers about whether it was a step too far
  - For example, a simple loop that seems “obviously safe” like:
    ```
    forall i in 1..n do
      A[i] = B[i]; // This is guaranteed to be safe since iterating over a range yields unique values, so no 'i's overlap
    ```
    - would need to become:
      ```
      forall i in 1..n with (ref A) do
        A[i] = B[i];
      ```
      - Or, it could be rewritten in a different style, like:
        ```
        forall (a, b) in zip(A, B) do // By iterating over the arrays, we can avoid the outer-scope access
          a = b;
        ```
DEFAULT INTENT CHANGES
Impact: larger forall-loops

- Larger simple loops similarly become more verbose without necessarily adding clarity, e.g.

```javascript
forall i in mySegInds
    with (var agg = new SrcAggregator(int)) {
        agg.copy(mySegs[i], segments[i]);
        agg.copy(myLens[i], lengths[i]);
        agg.copy(myELens[i], eLengths[i]);
        agg.copy(myESegs[i], encodeOffsets[i]);
    }

- might become:

```javascript
forall i in mySegInds
    with (var agg = new SrcAggregator(int),
             ref mySeqs, ref myLens, ref myELens, ref myESegs) {
        agg.copy(mySegs[i], segments[i]);
        agg.copy(myLens[i], lengths[i]);
        agg.copy(myELens[i], eLengths[i]);
        agg.copy(myESegs[i], encodeOffsets[i]);
    }
```
**DEFAULT INTENT CHANGES**

Impact: major forall-loops

- Meanwhile, already complicated loops become even more verbose, while perhaps adding some clarity
  - For example, the following 200+ line loop from Arkouda:

    ```java
    forall (i, column, ot, si, ui, ri, bi, segidx) in zip(0..#ncols, sym_names, col_objTypes,
            str_idx, int_idx, real_idx, bool_idx, segment_idx) do ...
    ```

  - Becomes:

    ```java
    forall (i, column, ot, si, ui, ri, bi, segidx) in zip(0..#ncols, sym_names, col_objTypes,
               str_idx, int_idx, real_idx, bool_idx, segment_idx)
    with (ref ptrList, ref segmentPtr, ref datatypes, ref sizeList,
           ref segarray_sizes, ref c_names, ref segment_tracking,
           ref str_vals, ref int_vals, ref real_vals, ref bool_vals) do ...
    ```
DEFAULT INTENT CHANGES

Next Steps

- Decide how we should handle the default ‘forall’ intent for arrays [#23488]
  - Use a default intent of ‘const’
    - Provides consistency with other cases
    - Draws attention to potential race conditions
    - Clearly identifies variables that are modified
    - Can make simple loops look overcomplicated
  - Preserve ‘ref if modified’ as the default intent
    - Makes it easy to write and parallelize code
    - Yet also makes it harder to visually detect potentially races
    - Preserves the current inconsistency

- Consider other mitigating strategies:
  - Improve the compiler’s ability to reason about modified variables
    - e.g., annotate iterators that yield unique indices to detect safe index sources
  - Add a blanket ‘ref’ intent that applies to many variables

```plaintext
coforall i in 1..10 with (ref A) do
  A[i] = i;
...
forall i in 1..10 do
  A[i] = i;

forall i in 1..10 (with ref *) ...
```

// deduce that all accesses of `A[i]` are safe
‘CONST’ INTENT IMPROVEMENTS
Background: Type-dependent intents

- Definitions of ‘const’ and default intents have depended on the type
  - The default intent has typically been ‘const’
- Here is a portion of the spec table describing the ‘const’ and default intents in Chapel 1.31:

<table>
<thead>
<tr>
<th>Type</th>
<th>const intent meaning</th>
<th>Default intent meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar types</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>(bool, int, uint, real, imag, complex)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-like types</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>(string, bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ranges</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>domains / domain maps</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>arrays</td>
<td>const ref</td>
<td>ref / const ref</td>
<td>see Default Intent for Arrays and Record ‘this’</td>
</tr>
<tr>
<td>records</td>
<td>const ref</td>
<td>const ref</td>
<td>see Default Intent for Arrays and Record ‘this’</td>
</tr>
</tbody>
</table>


We faced some problems with this approach to defining ‘const’ and default intents:
- table was hard to remember
- array accessors were hard to write
- constrained generics can’t have their intents vary based upon types
- larger integer sizes (e.g., ’int(4096)’) would require changing the intents to have reasonable behavior
- array performance is more sensitive to compiler optimization than would be ideal
‘CONST’ INTENT IMPROVEMENTS

This Effort: Formal and Task Intents

• Streamlined the language and improved optimization opportunities
  • redefined ‘const’ and default intents, including formal, return/yield, and task intents
  • spec change only: no compiler changes, yet

• ‘const’ now allows implementations to choose between ‘const ref’ and ‘const in’

• ‘const’ now asserts that, while the called function is running:
  • the value will not be changed through *indirect modification* (see example on the next slide)
    – for an ‘owned’ or ‘shared’ formal, it will not be modified to point to a different object
  • for an array formal, the array’s domain will not change

• It is the programmer’s responsibility to make sure that the above assertions are not violated
  • programs that violate the above assertions will not have the same behavior across different Chapel releases

• The default intent is usually ‘const’ and in that case carries the same assertions
  • default intent is now ‘const’ for all types except for ‘sync’ and ‘atomic’
‘CONST’ INTENT IMPROVEMENTS

This Effort: Indirect Modification

• A ‘const’ or ‘const ref’ formal, cannot be modified directly (because it is ‘const’)
• However, it’s still possible that the value is modified indirectly
• For example:

```plaintext
var a: int = 0;
f(a, a);
proc f(ref b, const ref c) {
    writeln(c);   // outputs 0
    a = 1;
    writeln(c);   // now outputs 1
    b = 2;
    writeln(c);   // now outputs 2
}
```

• ‘a = 1’ and ‘b = 2’ above are examples of indirect modification
• Indirect modification is legal for a ‘const ref’ formal, but not for a ‘const’ or default intent formal
  • not too surprising: ‘const’ means the implementation can choose between ‘const in’ or ‘const ref’
• Here is an example of an erroneous program:
  ```
  var D = {1..10};
  var A: [D] int;
  foo(A);
  proc foo(arr: []) {
    D = {1..3}; // assertion violated: array’s domain has changed
    A[1] = 3; // assertion violated: indirect modification of ‘array’
    ...
  }
  ```

• Programs that need this behavior should use ‘const ref’ intent to enable it
### ‘CONST’ INTENT IMPROVEMENTS

This Effort: Return and Yield intents

- Added an ‘out’ return and yield intent which returns/yields by value (the traditional default)
- Changed the ‘const’ return and yield intents in a similar manner to the formal intent
  - the implementation can choose between returning/yielding by value and by ‘const ref’
  - ‘const’ return and yield intent is currently specified but unstable

<table>
<thead>
<tr>
<th>intent</th>
<th>meaning for return</th>
<th>meaning for yield</th>
<th>current implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>‘out’</td>
<td>implementation-defined</td>
<td>by value (no change)</td>
</tr>
<tr>
<td>‘const’</td>
<td>implementation-defined</td>
<td>implementation-defined</td>
<td>by value (no change)</td>
</tr>
<tr>
<td>‘out’</td>
<td>by value</td>
<td>by value</td>
<td>by value</td>
</tr>
</tbody>
</table>
‘CONST’ INTENT IMPROVEMENTS
Impact and Next Steps

Impact:
- Addressed concerns with the design of ‘const’ intents
  - Intents have been simplified
- Programs that meet the new requirements of ‘const’ and default intent are stable

Next Steps:
- Add runtime checking of the new invariants that are implied by ‘const’/default intent, where possible
- Add optimizations that make use of the new properties
SPECIAL METHODS:
NAMING AND SAFETY
SPECIAL METHODS
Background

Chapel has a number of methods that have special behavior based on their names:

- Some have a role in the language’s definition, for example:

  ```chapel
  record R {
    proc init() { ... } // says how to create a new instance of ‘R’
    iter these() { ... } // says how to iterate over an instance of ‘R’ serially
  }
  ```

- Others have been used by features in the language and/or standard library, when defined:
  - And some of these cases, when not defined, would be provided by the compiler when possible
    ```chapel
    proc R.hash() { ... } // hash the object; used when creating associative domains, sets, or maps of ‘R’
    proc R.writeThis(f) { ... } // says how ‘R’ should be printed to a FileWriter
    ```

Special methods present a few challenges with respect to naming:

- How to make readers and authors of Chapel code more aware when these special methods are used?
- How to preserve the behavior of existing code if/when new special methods are added in the future?
  ```chapel
  proc R.foo() { ... } // if this is a user method, and special meaning is given to ‘foo( )’ later, ‘R’s interpretation would change
  ```
- How to avoid taking specific method signatures away from users?
Decided to use two techniques to demarcate special methods:

- **reserved keywords**
- **interfaces:**
  - Chapel has a start at supporting interfaces, similar to interfaces in Java et al., traits in Rust
  - existing support summarized in earlier release notes: [https://chapel-lang.org/releaseNotes/1.24/04-ongoing.pdf](https://chapel-lang.org/releaseNotes/1.24/04-ongoing.pdf) (unstable)

Reserved keywords were used for the following cases:

- initializer-related special methods
- default accessors / iterators

Interfaces were used for other cases:

- hashing
- IO serialization
- context management
KEYWORD-BASED SPECIAL METHODS
KEYWORD-BASED SPECIAL METHODS

This Effort

Decided to reserve the following special method names as keywords:

- initialization-related special methods:
  
  ```groovy
  proc init(...) ...
  proc init=(...) ... // this was already reserved prior to 1.32
  proc postinit() ...
  proc deinit() ...

  // keywords were also used as a replacement for ‘this.completeO;’ — see the section after the next
  ```

- default accessor/call and iterator methods:
  
  ```groovy
  proc this(...) ... // this was already reserved prior to 1.32
  iter these(...) ...
  ```
USING INTERFACES TO PROTECT SPECIAL METHODS
**INTERFACE-BASED SPECIAL METHODS**

This Effort

- Leverage interfaces to address problems with the current approach:
  - Associate a standard interface with each special method-based feature
  - To opt into a feature, the user datatype must explicitly *implement* its interface
  - If not opting in, the specially-named method is not treated specially
  - Special names added in the future and protected by interfaces will not affect existing code

```plaintext
interface hashable { // the standard interface for hashing
    proc hash(): uint;
}

record iImplement: hashable { // this record is declared to implement ‘hashable’, therefore...
    proc hash() { ... }     // this hash() method is used when hashing for associative domains, sets, etc.
}

record unRelated { // this hash() can be completely unrelated and do “unusual” things
    proc hash() { ... }
}
```
INTERFACES-BASED SPECIAL METHODS

This Effort

- Define standard interfaces to be recognized by the language and compiler

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Hashing</th>
<th>‘manage’ Statements</th>
<th>Serialization</th>
<th>Deserialization (via initializer)</th>
<th>Deserialization (via method)</th>
<th>Serialization + Deserialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>hash</td>
<td>enterContext, exitContext</td>
<td>serialize</td>
<td>init</td>
<td>deserialize</td>
<td>serialize, init, deserialized</td>
</tr>
</tbody>
</table>

- Stabilize only user-facing interface features required to support this change:
  - The syntax for implementing an interface:

    ```
    record myType: hashable { ... }
    class MyClass: ParentClass, contextManager { ... }
    ```

- Accepting user-defined methods with appropriate signatures as implementing an interface requirement
INTERFACE-BASED SPECIAL METHODS

Status

- In 1.32:
  - Existing code with user-defined special methods produces warnings about implementing the interface
    ```plaintext
type myRecord =
  proc hash() { … }  // warning: defining special method but not implementing its interface

record myRecord {
  proc hash() { … }
}
```

- Using a language feature without implementing its interface also issues a warning
  ```plaintext
type myRecord =
  proc hash() { … }  // warning is issued elsewhere

record myRecord {
  proc hash() { … }
}

var mySet: set(myRecord);  // warning: myRecord.hash() is used for built-in feature, but the interface isn’t implemented
```

- Standard interfaces are treated specially in order to:
  - Allow the return intent of a user’s `enterContext()` to be ‘ref’ or ‘const ref’ or ‘out’
  - Support the tricky genericity in `serialize()`, `init()`, `deserialize()`
INTERFACE-BASED SPECIAL METHODS

Next Steps

• Finalize the transition into interface-based special methods. After transition period:
  • Users will be able to define specially-named methods like hash() and serialize() that aren’t treated specially
    ```
    record myRecord1 {
      proc hash() { ... }
      proc hash() { ... }
    }
    // OK: this is a non-special method that happens to be named ‘hash’
    // compiler-generated special method is also provided
    ```
  • Using a language feature without implementing its interface will result in an error
    ```
    record myRecord2 {
      proc hash() { ... }
      proc hash() { ... }
    }
    // even when a non-special method is present
    // suppose the special method is not able to be compiler-generated
    var mySet: set(myRecord);  // error: trying to use myRecord for hashing, but it doesn’t implement hashable
    ```

• Stabilize the rest of interface features
• Improve design and implementation to support all special methods without special treatment
• Continue to support auto-generated procedures and interfaces for ’hash’, ‘serialize’, ‘deserialize’
UPDATING 'THIS.COMPLETE();'
**THIS.COMPLETE**

**Background**

- Initializers cannot use ‘this’ as an object until it has been completely initialized
  - e.g., no method calls, no passing ‘this’ to another subroutine, etc.
- To indicate that an object is completely initialized, ‘[this.]complete();’ has traditionally been used:
  - indicates that:
    - the compiler should initialize any remaining uninitialized fields
    - the object is complete and ready for use

```haskell
record R {
  var x, y: int;
  proc init() {
    this.x = 42;       // manually initialize ‘x’
    this.complete();   // make the compiler default-initialize ‘y’; indicate that ‘this’ is ready for use
    this.foo();        // call a method on ‘this’ (illegal prior to ‘this.complete()’)
    bar(this);         // pass ‘this’ to a procedure (illegal prior to ‘this.complete()’)  
  }
}
```

• ‘complete’ has felt like something of a special method
  • syntactically appears to be a normal method call (by design)
  • doesn’t rely on reserved words
  • doesn’t preclude users from creating their own methods named ‘complete( )’
    – which can lead to confusion if a type happens to define a 0-argument method named ‘complete( )’

• However, we didn’t want to reserve ‘complete’
  • it felt too frequently used and useful in code to reserve for this very specific purpose

• It also didn’t fit the pattern of using an interface
  • not something the user needs to define on their types, unlike other interface-based special methods
  • more of a special hook in the language or compiler than a normal method

• For this reason, decided to look at alternate designs, focusing primarily on keyword-based notations

• Ended up deciding to use ‘init this;’
  • keyword-based
  • reasonably mnemonic
Impact

• Removes another special method-like case from concern
• Relies on existing keywords
• Uses a less-traditional syntactic pattern to make this fairly unique feature stand apart better

```java
record R {
    var x, y: int;
    proc init() {
        this.x = 42;  // manually initialize 'x'
        init this;    // make the compiler to initialize 'y'; indicate that 'this' is ready for use
        this.foo();  // call a method on 'this' (illegal prior to 'init this;')
        bar(this);   // pass 'this' to a procedure (illegal prior to 'init this;')
    }
}
```
Next Steps

- Consider relaxing the need to write ‘init this;’ when the compiler can determine where it goes [#22975]

- When the object is obviously fully initialized:
  ```java
  record R {
    var x, y: int;
    proc init() {
      this.x = 42; // manually initialize ‘x’
      this.y = 45; // manually initialize ‘y’
      // the compiler knows that all fields are initialized here, so could implicitly insert ‘init this;’
      this.foo(); // call a method on ‘this’
    }
  }
  ```

- Or maybe even when it’s not?
  ```java
  proc R.init() {
    this.x = 42; // manually initialize ‘x’
    // this is obviously the transition from initialization to use, so should the compiler insert ‘init this;’ here as well?
    this.foo();
  }
  ```
OTHER IDENTIFIER-RELATED CHANGES
'SUPER' AND 'RANGE' AS RESERVED WORDS

Background:

- Reserved words may not be used as identifiers for user-written variables, subroutines, types, etc.
- Prior to this release, 'range' and 'super' were not reserved
  - 'range' is a fundamental datatype
  - 'super' is a keyword used to access members of an object’s superclass
- These are similar in nature to other reserved words we have (such as 'locale' or 'this')

This Effort: Made 'super' and 'range' reserved words

Impact: We now produce this error if they are used as identifiers:

// The following:
var super, range;

// Now errors out with:
error: attempt to redefine reserved word 'super'
error: attempt to redefine reserved word 'range'
DEPRECATE USING '$' IN IDENTIFIERS

Background:
- 'sync'/single' variables used to support implicit, blocking, reads/writes
- Previously, by convention, we suggested suffixing these variables with ‘$’ to flag the potential for dead/livelock
- Now that implicit reads/writes are no longer supported, this convention is not as important

This Effort:
- We decided to disallow the use of ‘$’ within identifiers via a deprecation warning
  - simplifies the language
  - reserves ‘$’ as a token for potential future use

Impact:
- We now produce this warning:

  warning: Using '$' in identifiers is deprecated; rename this to not use a '$'

Next Steps: Deprecate support for ‘$’ altogether, barring objection from users
DEPRECATED BEHAVIORS FOR GENERIC NUMERIC ARGUMENTS
GENERIC NUMERIC ARGUMENTS

Background

- Using the argument query syntax, users can write routines that accept varying numeric widths
  - E.g., this routine accepts pairs of integers of various widths:
    
    ```
    proc add(x: int(?w), y: int(w)) { return x + y; }
    ```

- Historically, the compiler allowed implicitly converting from other primitive types in such cases:
  - ```
    add(true, true);  // since 'bool' implicitly converts to '1', the compiler implicitly converted each 'true' to '1', resulting in '2'
    ```

- However, this behavior was inconsistent with the normal treatment of other generic formals
  - E.g., consider 'numeric', a generic type that can be instantiated with any 'int'/'uint'/'imag'/'real'/'complex':
    
    ```
    proc addNumeric(x: numeric, y: x.type) {
        return x + y;
    }
    ```
    ```
    addNumeric(true, true);  // fails to compile: candidate does not match since 'true'/bool is not a 'numeric' type
    ```
The behavior on the previous slide is due to how the compiler implements generic numeric arguments.

Given:

```java
proc add(x: int(?w), y: int(w)) { return x + y; }
```

The compiler generates the following instantiations of `add()`, as implied by legal values of `w`:

```java
proc add(x: int(8), y: int(8)) { return x + y; }
proc add(x: int(16), y: int(16)) { return x + y; }
proc add(x: int(32), y: int(32)) { return x + y; }
proc add(x: int(64), y: int(64)) { return x + y; }
```

This approach is odd for a few additional reasons:

- It’s only possible because types like `int` are built-in, permitting the compiler to know all legal instantiations.
  - In contrast, if the formal was a generic record query, the compiler could not proactively stamp out all legal overloads.
- It supports calls like the following, where a literal reading of the signature makes this seem inappropriate:
  ```java
  add(myInt32, myInt64); // ‘w’ would be inferred as ‘32’, constraining ‘y’ to ‘int(32)’; so ‘myInt64’ should not be a legal actual
  ```
  - This resolved to the `int(64)` overload due to the “stamping out” implementation.
DEPRECATED CONVERSIONS
This Effort

- Deprecated implicit conversions for formals using types like ‘int(?w)’
  - E.g., given a routine like the following defined in a user module:
    ```ruby
    proc add(x: int(?w), y: int(w)) { return x + y; }
    ```
  - The compiler now produces a warning for calls like these:
    ```ruby
    add(true, true);  // warning: deprecated use of implicit conversion when passing to a generic formal
    add(myInt32, myInt64);  // warning: deprecated use of implicit conversion when passing to a generic formal
    ```

- Rewrote key parts of the standard library to stop relying on the “stamping out” behavior
  - e.g., math operators like +, −, ∗, /, … were changed from numeric queries to explicit instantiations
DEPRECATED CONVERSIONS

Next Steps

- Remove the special handling for generic numeric arguments, treating them like typical generic queries
- Review remaining uses of generic numeric arguments in the standard library
  - if implicit conversions are appropriate for a routine, create overloads for each concrete type (as with math ops)
  - otherwise, no action is needed
LIFETIME OF NESTED CALL EXPRESSION TEMPORARIES
LIFETIME OF TEMPORARIES

Background

- The compiler introduces temporary variables for nested call expressions:

  ```
  f(g());
  // compiler translates it into
  var tmp; tmp = g(); f(tmp); deinitialize(tmp);
  ```

- When to deinitialize such a temporary depended on whether it was contained in an initialization:

  ```
  f(g()); // end of statement because it’s not initializing a variable
  var x = f(g()); // end of block because it’s initializing ‘x’
  const ref y = f(g()); // end of block because it’s initializing ‘y’
  ```

- However, users have indicated that they find the current rules confusing due to split-init:

  ```
  var z;
  z = f(g()); // end of block because this is a split initialization of ‘z’
  z = f(g()); // end of statement because this one is an assignment to ‘z’, not initialization
  ```

- Concern: these two cases look the same but behave differently
LIFETIME OF TEMPORARIES
This Effort and Impact

This Effort: Made temporaries initializing a ‘var’ or ‘const’ use end-of-statement rule

- Now deinitialization point for a temporary is end-of-statement unless it is used to initialize a ‘ref’ or ‘const ref’

```cpp
f(g());       // end of statement (as before)
var x = f(g()); // end of statement (new)
const ref y = f(g()); // end of block (as before)
```

- Kept the end-of-block behavior for initializing a ‘ref’ or ‘const ref’ to support array slices

Impact:

- Removed a source of confusion for ‘const’ and ‘var’ declarations
L-VALUE CHECKING IMPROVEMENTS
L-VALUE IMPROVEMENTS

Background

• An ‘l-value’ is a value that can be modified with a location where the modification can persist
  
  ```
  var x: int;
  x = 1;       // ok: ‘x’ is an l-value
  (x+1) = 3;   // error: ‘x+1’ is not an l-value, so the assignment to ‘3’ here would be lost
  ```

• Compilers use l-value errors to reduce confusion around compiler-introduced temporaries
  
  • the assignment to ‘(x+1)’ above sets a temporary that will be lost
  • so, it probably represents a programmer error

• 1.31 had some inconsistent l-value errors:
  
  ```
  (A+1) = 3;       // l-value error
  makeA() = 3;     // not an error -- but arguably confusing
  fConstRef(1);    // l-value error -- but not confusing
  ```

  ```
  var A: [1..10] int;
  proc makeA() { return [1,2,3]; }
  proc fConstRef(const ref arg) { }
  ```
**L-VALUE IMPROVEMENTS**

This Effort and Impact

**This Effort:** Improved two cases where the l-value checking had a surprising result

- Removed exception to l-value checking for arrays to make the language more consistent
  - it still applies to array slices since they are more like references than values
- Enabled passing a numeric param or literal to a ‘const ref’ formal without generating an l-value error
  - since there is no chance for confusion as described above with this case

**Impact:** l-value checking is now more consistent

```
var A: [1..10] int;
proc makeA() { return [1,2,3]; }
proc fConstRef(const ref arg) { }

(A+1) = 3; // still an l-value error
makeA() = 3; // now an l-value error in 1.32
fConstRef(1); // now legal
```
TYPE-RELATED CHANGES

- Array Creation and OOM
- Range Stabilization
- Domain Stabilization
- Array Stabilization
- String/Bytes Changes
  - c_string Deprecation
- Sync / Atomic Stabilization
- Class Stabilization
THROWING ARRAY CREATION API FOR OUT-OF-MEMORY ERRORS
THROWING ARRAY CREATION INTERFACE

Background

- Users were reporting running out of memory while using Arkouda
  - All work is lost when the server crashes due to running out of memory
  - Lots of time-consuming effort can be lost when a long-standing server crashes

- Would like to be able to gracefully handle errors, rather than crash the server

- This is a challenging problem due to overcommit, fragmentation, undefined behavior, etc.

- Decided to implement an array creation interface that will throw if out of memory
  - This means that non-array allocations that exceed the memory limit may still crash the server
  - Large array allocations use most memory in Arkouda, making them the most likely culprit

- Implementation in the 1.32 Chapel release is limited to fixed-heap configurations
  - Other configurations may not detect out-of-memory conditions due to overcommit
THROWING ARRAY CREATION INTERFACE

Impact

- Throwing interface available as a method on Block and default rectangular domains
- Standard array creation:
  ```javascript
  const dom = blockDist.createDomain(0..#size);
  var A: [dom] int;
  ```

- Throwing array creation:
  ```javascript
  const dom = blockDist.createDomain(0..#size);
  try {
    var A = dom.tryCreateArray(int);
  } catch e {
    // continue execution with the knowledge that the array was not created
  }
  ```

- A throwing ‘tryCopy( )’ method on arrays is also supported
Next Steps

• Investigate 2 performance optimizations for the new interface:

  1. Skip default initialization of arrays using this interface when given an initialization expression
     – Currently will always default initialize arrays
     – This could improve performance for operations like copying arrays

  2. Add support for array stealing on arrays created with this interface
     – Generally, when an array is used in an assignment, if it is not used again, the LHS array can steal the RHS array
     – The compiler currently only supports array stealing on arrays created using the standard Chapel array syntax

• Investigate extending the interface to work for non-fixed-heap configurations

• Consider other approaches to handling OOM conditions within the language and library
  • Introduce an ‘unchecked error’ concept that need not be caught, but can be?
  • Introduce distributions that are known to ‘throw’ on OOM? (e.g., ‘throwingBlockDist’)
RANGE TYPE STABILIZATION
RANGE STABILIZATION

Background and This Effort

**Background:**

- ‘range’ is a predefined type for constant-space representation of regular sequences of values
- We felt some properties of ranges could be improved
  - naming, definitions, functionality

**This Effort:**

- Changed naming and fine-tuned the definitions related to range bounds, stride, and alignment
- Made the static type more precise w.r.t. the strides it allows
- Clarified some operations on unbounded ranges over ‘bool’ and enums
- Other changes are discussed below and listed in CHANGES.md
  - casts, literals with mixed-type param bounds, unstable and deprecated features, etc.
**RANGE STABILIZATION**

Range Bounds, Strides, Alignment: Summary

The following naming and semantic improvements are discussed next:

<table>
<thead>
<tr>
<th>previously</th>
<th>in 1.32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>param range.boundedType</strong> : BoundedRangeType</td>
<td><strong>param range.bounds</strong> : boundKind</td>
</tr>
<tr>
<td><strong>enum BoundedRangeType</strong> { bounded, boundedLow,</td>
<td><strong>enum boundKind</strong> { both, low,</td>
</tr>
<tr>
<td>boundedHigh, boundedNone }</td>
<td>high, neither }</td>
</tr>
<tr>
<td><strong>param range.stridable</strong> : bool</td>
<td><strong>param range.strides</strong> : strideKind</td>
</tr>
<tr>
<td><strong>proc range.aligned</strong></td>
<td><strong>proc range.isAligned()</strong></td>
</tr>
<tr>
<td><strong>proc range.isAmbiguous()</strong></td>
<td><strong>deprecated</strong></td>
</tr>
<tr>
<td>unambiguous alignment can be any index</td>
<td>0 &lt;= alignment &lt; abs(stride)</td>
</tr>
</tbody>
</table>
RANGE STABILIZATION

Range Bounds: Details

Changed names, kept existing semantics

- The new names are shorter, easier to use

```c
/* old enum: */
enum BoundedRangeType { bounded, boundedLow, boundedHigh, boundedNone }

/* new enum: */
enum boundKind { both, low, high, neither }

/* old field: */
param range.boundedType: BoundedRangeType = BoundedRangeType.bounded;

/* new field: */
param range.bounds: boundKind = boundKind.both;
```
Renamed the field; changed its type from ‘bool’ to an enum

- The new field name is shorter, clearer, consistent with the field ‘bounds’
- Enum carries more information, enables optimizations in more cases
  - compiler knows that these range’s strides are positive: ‘1..n by 2’, ‘1..n by s: uint’, ‘r: range(strides=strideKind.positive)’

```cpp
/* old field: */  param range.stridable: bool = false;  // ‘false’ implies ‘stride’ is always 1
/* new field: */  param range.strides:  strideKind = strideKind.one;
/* new enum: */   enum strideKind {  // enum values constrain ‘stride’ as follows:
    one,       // stride == 1
    negOne,    // stride == –1
    positive,  // stride > 0
    negative,  // stride < 0
    any }      // stride != 0

// for example, ‘r1.stride’ is known at compile time; ‘r2.stride’ can be any negative integer
var r1: range(strides=strideKind.one);      ...; compilerAssert(r1.stride == 1);
var r2: range(strides=strideKind.negative); ...; assert(r2.stride < 0);
```
### RANGE STABILIZATION

#### Range Alignment: Details

Revised definitions in 1.32:

- Alignment, when unambiguous, is always within $0 \ldots |\text{stride}| - 1$
  - stride == 1 or -1 always implies alignment == 0
  - alignment given by user is stored mod($|\text{stride}|$)

  ```
  const r = 1..n by 2 align 3;
  writeln(r.alignment); // prints 1
  ```

- Ranges over ‘bool’ and enum types are always aligned
- Range is “aligned” IFF its alignment is unambiguous
  - these properties used to differ for unstridable ranges, introducing unnecessary cognitive and coding complexity

- Renamed ‘range.aligned’ to ‘range.isAligned( )’
  - the new name is clearer

- Deprecated ‘range.isAmbiguous( )’
  - avoids two ways of obtaining the same information: ‘r.isAmbiguous( )’ is the same as ‘! r.isAligned( )’
**RANGE STABILIZATION**

Unbounded Ranges over `bool` and Enums

Any sequence of indices for a range over `bool` or enum is finite

- E.g., `var r = false .. ;` means `r` represents the sequence `[false, true];` likewise, when the index type is an enum

In 1.32, range operations treat a range with a missing bound...

...as unbounded, when inquiring about that bound using `hasLowBound()`, `low`, `lowBound`, or `...high...`

...as bounded, for most operations related to the index sequence

For example, given:

```plaintext
enum color { red, yellow, green }; const r = color.red..;
```

- treat `r` as having no high bound:
  ```plaintext
  r.hasHighBound(); // false
  r.high; // error: no high bound
  r.highBound; // 
  ```

- the following are conservatively disallowed for now:
  ```plaintext
  r.size; // error: `r` is not bounded
  var colorDomain = { r }; // 
  ```

- treat `r` as having the high bound `color.green`:
  ```plaintext
  for c in r do // iteration behavior is stable
    writeln(c); // prints: red yellow green
  r.hasLast(); // unstable; true
  r.last; // unstable; color.green
  r.isEmpty(); // unstable; false
  write(r == color.red..color.green); // unstable; true
  ```
RANGE STABILIZATION
Cast, safeCast(), tryCast() in 1.32

• Range cast `:` ensures that the source range’s stride is suitable for the target type
  – Halts upon violations; behavior is undefined if compiled with '--fast' or '--no-checks'
  – E.g., the following halts because the stride 256 does not fit in int(8) and is not acceptable for strideKind.negative
    ```
    var s = 256;
    var r = ( 1.. by s ) : range(int(8), boundKind.low, strideKind.negative); // halts
    ```
  – Cast(s) of the bound(s) to the target idxType are unchecked, for convenience
    ```
    var r = (1..99): range(int(8)); // OK
    ```
•Introduced ‘range.tryCast()’, which ‘throws’ upon any violation
  ```
  var r = (1..256).tryCast(range(int(8))); // throws an IllegalArgumentException
  ```
• Deprecated ‘range.safeCast()’, which halts upon violations
  – Suggested replacement: ‘range.tryCast()’
• Range cast `:` and ‘tryCast’ are unstable when source and/or target ‘idxType’ is an enum type
  ```
  enum E { a = 1, b = 10 }
  var r = (E.a .. E.b) : range(int); // should 'r' be 0..1 (preserves the size of E.a .. E.b)
  // or 1..10 (preserves the numeric values; this is the behavior in 1.32) ?
  ```
RANGE STABILIZATION

‘idxType’ of Range Literals with Mixed-Type Param Bounds

We want the index type of a range literal with mixed-type bounds to match the behavior of ‘+’

- For example, the type of ‘1 + 31:int(8)’ is int(64), so we want the index type of ‘1 .. 31:int(8)’ to be int(64), too
- Previously range literals with mixed-type param bounds were treated differently

In 1.32, started the transition to the matching behavior:

- By default, use the previous rule, generate a warning
- When compiled with ‘-snewRangeLiteralType’, use the desired rule, no warning
- Ultimately, the current behavior with ‘-snewRangeLiteralType’ will be the default and the option will be removed
- For example:

<table>
<thead>
<tr>
<th>idxType of:</th>
<th>in 1.31</th>
<th>in 1.32</th>
<th>in 1.32, if compiled with -snewRangeLiteralType</th>
<th>stabilization target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .. 31:int(8)</td>
<td>int(8)</td>
<td>int(8)</td>
<td>int(64)</td>
<td>int(64)</td>
</tr>
</tbody>
</table>
RANGE STABILIZATION
Impact, Status, and Next Steps

**Impact:** improved semantics and ergonomics of ranges

**Status:** most range features are stable

**Next Steps:** stabilize the remaining features based on user feedback and need
DOMAIN TYPE STABILIZATION
Background and This Effort

**Background:**
- Domains represent index sets and iteration spaces, and are a building block for Chapel arrays
- A variety of design questions have been raised as part of 2.0 stabilization work

**This Effort:**
- Conducted multiple design reviews
- Stabilized many existing features as-is, with stronger rationales
- Most `range` changes apply dimension-wise to rectangular domains
- These changes are discussed elsewhere in release notes:
  - marked associative and sparse domains as unstable
  - now require writing the generic domain type as ‘domain(?)’
- Other changes are discussed below and listed in CHANGES.md
  - casts, `localSubdomain()`, unstable and deprecated features, etc.
DOMAIN STABILIZATION
Domain cast, safeCast(), tryCast()

In 1.32, cast operations on rectangular domains perform dimension-wise range casts

- Domain cast ‘:’ ensures that the source domain’s strides are suitable for the target type
  - Halts upon violations; behavior is undefined if compiled with ‘--fast’ or ‘--no-checks’

- Introduced ‘domains.tryCast()’
  - ‘throws’ upon any violation

- Deprecated ‘domain.safeCast()’, which halts upon violations
  - Suggested replacement: ‘domains.tryCast()’

- Domain cast ‘:’ and ‘tryCast’ are unstable when source and/or target idxType is an enum type
**DOMAIN STABILIZATION**

localSubdomain(), localSubdomains(), hasSingleLocalSubdomain()

In 1.32, made these changes:

- ‘domain.localSubdomain( )’ is now stable for Block and Cyclic distributions
  - Supports users of these distributions

- Marked ‘domain.hasSingleLocalSubdomain( )’ and ‘iter domain.localSubdomains( )’ as unstable
  - Unclear whether these should be available by default and whether ‘hasSingleLocalSubdomain’ should be ‘param’
ARRAY TYPE STABILIZATION
ARRAY STABILIZATION
This Effort

Array changes in 1.32 include:

• An array returned by value cannot be passed to a ‘ref’ formal
  – removes an unnecessary exception from the general rule
    ```
    proc createArray() out { return someArray; }
    proc processArray(ref A) { ... }
    processArray(createArray());  // error: non-lvalue actual is passed to ‘ref’ formal
    ```

• These changes are discussed elsewhere in release notes:
  – introduced array creation interface that throws if out of memory
  – the domain of a ‘const’ array formal now must remain unchanged during the call

• Other changes are listed in CHANGES.md
  – enabled promotion of cast ‘:’
  – ‘readBinary( )’ and ‘writeBinary( )’ now support multi-dimensional arrays
  – enabled array slicing with unaligned ranges
  – unstable and deprecated features, etc.
STRING & BYTES CHANGES
STRING & BYTES

Background and This Effort

**Background:**
- The 'string' and 'bytes' types have similar interfaces
- However, the factory methods were awkward to use in generic code handling both 'string' and 'bytes', e.g.,
  ```java
  if t == string then createStringWithNewBuffer(...) else createBytesWithNewBuffer(...);
  ```

**This Effort:**
- Converted the factory methods for 'string' and 'bytes' to type methods & improved clarity of names
- Marked the '.createBorrowingBuffer(…)' methods unstable because they may perform a copy

<table>
<thead>
<tr>
<th>Previous Factory</th>
<th>Replacement Factory</th>
</tr>
</thead>
<tbody>
<tr>
<td>createStringWithOwnedBuffer()</td>
<td>string.createAdoptingBuffer()</td>
</tr>
<tr>
<td>createStringWithBorrowedBuffer()</td>
<td>string.createBorrowingBuffer()</td>
</tr>
<tr>
<td>createStringWithNewBuffer()</td>
<td>string.createCopyingBuffer()</td>
</tr>
<tr>
<td>createBytesWithOwnedBuffer()</td>
<td>bytes.createAdoptingBuffer()</td>
</tr>
<tr>
<td>createBytesWithBorrowedBuffer()</td>
<td>bytes.createBorrowingBuffer()</td>
</tr>
<tr>
<td>createBytesWithNewBuffer()</td>
<td>bytes.createCopyingBuffer()</td>
</tr>
</tbody>
</table>
Impact and Next Steps

Impact:
• Code that previously required a conditional check on the type can now be expressed in a simpler way, e.g.,

```
if t == string then createStringWithNewBuffer(...) else createBytesWithNewBuffer(...);
```

can be simplified to:

```
t.createCopyingBuffer(...);
```

Next Steps:
• Stabilize `createBorrowingBuffer(....)` for 'string' and 'bytes'
C_STRING DEPRECIATION
C_STRING DEPRECATION
Background and This Effort

Background:
- The 'c_string' type mapped to a 'const char*' in C, and provided interop for string data between Chapel and C
- 'c_string' remained in the language for historical reasons, yet we felt that:
  - it was an odd outlier since C doesn't have strings
  - it created confusion around memory management, e.g.,
    ```
    var x = myString.c_str();  // x does not need to be freed, it just aliases the data
    ```

This Effort: Deprecated 'c_string' type in favor of 'c_ptrConst(c_char)'
- Replaced 'c_string' with 'c_ptrConst(c_char)' in user interfaces
- Moved '.'c_str()' on string and bytes into the 'CTypes' module
- Added an unstable 'strLen()' procedure to the 'CTypes' module as a replacement for 'c_string.size'
C_STRING DEPRECATION
Impact and Next Steps

**Impact:** Code that used 'c_string' should be updated to use 'c_ptrConst(c_char)'

- The following 'c_string' methods are deprecated without replacement:
  - `writeThis()`, `serialize()`, `readThis()`, `indexOf()`, `substring()`
- Some special handling may be required to update 'c_string' to 'c_ptrConst(c_char)', especially for casting

<table>
<thead>
<tr>
<th>if your code is...</th>
<th>the recommendation is to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>casting 'c_string' to 'string' or 'bytes'</td>
<td>use appropriate type's <code>.createCopyingBuffer()</code> factory</td>
</tr>
<tr>
<td>casting 'string' or 'bytes' to 'c_string'</td>
<td>replace the cast with '.c_str()'</td>
</tr>
<tr>
<td>casting 'c_string' to other types, e.g., 'int'</td>
<td>use 'string.createCopyingBuffer()'; then cast</td>
</tr>
<tr>
<td>casting other types to 'c_string'</td>
<td>convert the other type to a 'string' and call '.c_str()'</td>
</tr>
<tr>
<td>using a 'param c_string'</td>
<td>use a 'param string' instead</td>
</tr>
</tbody>
</table>

**Next Steps:**
- Decide which, if any, convenience functions to include for 'c_ptr'/c_ptrConst(c_char)' [#21052]
- Continue to remove 'c_string' from the compiler and fully remove it from the language
SYNC / SINGLE / ATOMIC STABILIZATION
DEFAULT INITIALIZING
SYNCS AND ATOMICS
Background

- Chapel supports ‘sync’ and ‘atomic’ as synchronization types
- Implicit reads/writes on ‘sync’s were deprecated in Chapel 1.24, requiring explicit calls to match ‘atomic’
- The compiler generates default initializers for aggregate types like ‘class’ or ‘record’

```chapel
class MyClass {
  var x: int;
  var y: sync int;
}
proc MyClass.init(x: int = 0, y: sync int = ...) { // compiler-generated
  this.x = x;
  this.y = y; // requires implicit reads and writes
}
```

- By default, ‘sync’ and ‘atomic’ are by passed by ‘ref’ and returned by value

```chapel
var myGlobalSync: sync int;
proc getSync(s: sync real) do // this passes ‘s’ by ‘ref’
  return myGlobalSync; // this makes a copy
```
This Effort:

- Deprecated compiler-generated initializers for a ‘class’ or ‘record’ when it contains ‘sync’ and ‘atomic’ fields
  - Such compiler-generated initializers rely upon implicit reads and writes
  - It also seems unusual to initialize one ‘sync’/‘atomic’ with another implicitly
- Deprecated returning a ‘sync’ or ‘atomic’ by value
  - Copying a synchronization type relies upon implicit reads and writes
  - Copying a synchronization type has questionable value

Next Steps:

- Consider whether to re-enable these features in a different form:
  - A default initializer for a ‘sync t’/‘atomic t’ field could take a ‘t’ argument and use the standard copy initializer?
  - Perhaps the default return intent for ‘sync’/‘atomic’ types should be ‘ref’?
    - would match the asymmetry in their argument intents
- Remove deprecated support for implicit reads/writes of ‘sync’
SYNC/SINGLE STABILIZATION
SYNC/SINGLE STABILIZATION

**Background:** Sync/single variables provide a mechanism to coordinate between tasks

- Have a full-empty bit (FEB) state, modeled after Tera MTA / Cray XMT and predating atomics
  - Blocking methods will suspend the current task if the variable is in the wrong full/empty state
    - `readFE()`, `writeEF()`, `readFF()`
  - Non-blocking methods ignore state and just provide atomicity
    - `readXX()`, `writeXF()`, `reset()`, `isFull`

- Single variables are a specialization of sync variables that can only be written once

**This Effort:** Made the non-blocking API unstable and deprecated single variables

- Parts of non-blocking API are only safe in serial regions
  - Either want to expand to support concurrent non-blocking access or deprecate in the longer term
- Single variables have minimal benefit since implicit reads were deprecated
  - Same functionality available with syncs, write-once difference does not seem worth keeping the additional type
ATOMIC STABILIZATION
**ATOMIC STABILIZATION**

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

- Additionally, Chapel includes ‘compareAndSwap(expected, desired)’
  - Similar to ‘compareExchange(ref expected, desired)’, except ‘expected’ is not passed by reference or updated on failure
  - There are use-cases that do not want to update ‘expected’
    - But behavior difference is not obvious just based on the names

**This Effort:** Marked ‘compareAndSwap( )’ unstable

- Expect functionality to remain in some form, but may change names
RENAMEING ‘OBJECT’ TO ‘ROOTCLASS’
Background:

- Chapel defines a root class type from which all other classes inherit
  ```chapel
class C { … } // class C names no parent class type, so it is a child of the root class
class D: C { … } // class D is a child class of C, so it also inherits from the root class, just indirectly
  ```
- Traditionally, Chapel has called this root class ‘object’
  - follows the precedent set by many other OO languages
- However, this name increasingly gave us pause, because...
  - class types in Chapel typically use upper-case names, suggesting ‘Object’ would be the more appropriate capitalization
  - we refer to both record and class instances as “objects”, making either capitalization a bit more ambiguous than ideal

This Effort: Deprecated ‘object’, renaming it to ‘RootClass’

- follows our preferred capitalization scheme
- distinguishes itself better as being specific to classes
- improves accuracy by focusing on it being a type (“…Class”) rather than a value (“object”/“…Object”)

Next Steps: Remove support for ‘object’ after it’s been deprecated for a few releases
CLASS MANAGEMENT UPDATES
Background and This Effort

Background: Chapel supported multiple disparate ways to convert between class management styles

This Effort:

- Deprecated some value-based ways to convert management styles in favor of using `.adopt()` and `.release()`
  - Deprecated `.create()`, `.retain()`, and `.clear()`
  - Deprecated casting from `owned` to `shared`
  - Deprecated casting to `unmanaged`

```chapel
var myOwned = new MyClass?(); // ‘myOwned’ is a nilable owned object
var myShared = shared.adopt(owned.release(myOwned)); // ‘myShared’ is a nilable shared object
// ‘myOwned’ is now ‘nil’, ‘myShared’ manages the lifetime of the object

// myShared2 is a non-nilable shared object, it participates in the lifetime management of the object
var myShared2 = myShared;

myShared = nil; // myShared no longer participates in the lifetime of the object
writeln(myShared2); // myShared2 can still be used
```
This Effort (continued):

- Deprecated the ‘owned.borrow( )’ type method for changing management types
- Fixed usages of a type cast to ‘borrowed’ on argument types

```pascal
type myOwnedType = owned MyClass;

type myBorrowedType = myOwnedType: borrowed;

proc myFunction(x: myBorrowedType, y: myOwnedType: borrowed) do
  writeln("x.type = ", x.type:string, ", y.type = ", y.type:string, ",");

var a = new myOwnedType();
var b = new (myOwnedType:shared) (); // b is a shared MyClass
myFunction(a, b.borrow()); // implicit or explicit borrowing
```

Status: ‘owned’ and ‘shared’ are now stable

Next Steps: Improve the error messages for ‘owned’ and ‘shared’
OTHER LANGUAGE IMPROVEMENTS
OTHER LANGUAGE IMPROVEMENTS

For a more complete list of language changes and improvements in the 1.31 and 1.32 releases, refer to the following sections in the CHANGES.md file:

- New Language Features
- Language Feature Improvements
- Syntactic / Naming Changes
- Semantic Changes / Changes to the Chapel Language
- Unstable Language Features
- Deprecated / Removed Language Features
- Language Specification Improvements
- Other Documentation Improvements
- Example Codes
- Bug Fixes
THANK YOU

https://chapel-lang.org
@ChapelLanguage