Hewlett Packard
Enterprise

CHAPEL 1.25 RELEASE NOTES: LANGUAGE IMPROVEMENTS

Chapel Team

September 23, 2021

OUTLINE

- Foreach Loops
- Operator Improvements
- Manage Statement
- Resizing Non-Nilable Arrays
- Ranges / Domains / Arrays: API Changes
- Slicing Associative Arrays
- Ranges and Single-Value Enums
- Improvements to Interfaces
- Other Language Improvements

FOREACH LOOPS

FOREACH LOOPS
Background and This Effort

Background

- ‘foreach’ loops were proposed in the 1.24 timeframe as a new kind of loop
  – Signifies that the loop body is order-independent
  – Parallelizable but does not create new tasks
  – Implies that the loop body is safe for vectorization or being launched as a GPU kernel

This Effort

- In 1.25, ‘foreach’ loops are implemented

```
foreach i in a.domain do    // this loop is order-independent
  a[i] += i*2;
```

FOREACH LOOPS

Impact

```
iter myIter() {
  for ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable

Neither the loop nor the iterator uses 'foreach'

FOREACH LOOPS

Impact

```
iter myIter() {
  for ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable

Neither the loop nor the iterator uses 'foreach'

```
iter myIter() {
  foreach ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable

The loop does not use 'foreach'

FOREACH LOOPS
Impact

```
iter myIter() {
  for ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable
Neither the loop nor the iterator uses ‘foreach’

```
iter myIter() {
  for ... do
    yield ...;
}

foreach i in myIter() do   // or ‘forall’
  a[i] = i*2;
```

Not Vectorizable
The iterator does not use ‘foreach’

```
iter myIter() {
  foreach ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable
The loop does not use ‘foreach’

FOREACH LOOPS

Impact

```
iter myIter() {
  for ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable

Neither the loop nor the iterator uses 'foreach'

```
iter myIter() {
  for ... do
    yield ...;
}

foreach i in myIter() do   // or 'forall'
  a[i] = i*2;
```

Not Vectorizable

The iterator does not use 'foreach'

```
iter myIter() {
  foreach ... do
    yield ...;
}

for i in myIter() do
  a[i] = i*2;
```

Not Vectorizable

The loop does not use 'foreach

```
iter myIter() {
  foreach ... do
    yield ...;
}

foreach i in myIter() do   // or 'forall'
  a[i] = i*2;
```

Vectorizable

Both the loop and the iterator use 'foreach'

FOREACH LOOPS

Status and Next Steps

Status

- ‘foreach’ is a new loop type that can potentially parallelize the loop through vectorization or GPU kernel launch
- ‘forall’ loops already indicated order-independence of the loop body
  - So, they are generally vectorizable
  - As with ‘foreach’, follower iterator(s) need to use ‘foreach’ to indicate order-independence
- ‘foreach’ loops do not support intents yet
- Outer-loop vectorization is not yet supported

Next Steps

- Enable outer loop vectorization
  - We are planning to incorporate Region Vectorizer in the bundled LLVM to enable outer-loop vectorization
- Implement intents for ‘foreach’ to enable:
  - vector-lane-private variables
  - vectorized reductions
  - See issue #18500


OPERATOR IMPROVEMENTS

OPERATORS

Background and This Effort

Background:

- Operators have traditionally been defined as standalone functions with symbolic names

```
proc +(a: Foo, b: Bar) { … }
```

  – Standalone functions were complicated to make visible in all scopes in which the operators were used

- In the 1.24 release, added support for declaring operators with an ‘operator’ keyword

```
operator +(a: Foo, b: Bar) { … }
```

- Also enabled declaring operators as methods

```
operator Foo.+(a: Foo, b: Foo) { … }
```

  – Operator methods are treated like normal methods—primary and secondary operators are available to all instances

This Effort:

- Deprecated declaring operators with ‘proc’, must now use ‘operator’ keyword form
- Made the compiler generate default operator methods instead of standalone operators for types
- Updated libraries to use operator methods instead of standalone operators when applicable

OPERATORS

Impact and Next Steps

Impact:

- All operators can be found by searching for ‘operator’ in code
- Default operators for a type are guaranteed to be callable on any instance
  - Even if the type declaration is not otherwise visible to the current scope, as with methods

Next Steps:

- Add support for forwarding operator methods
- Enable listing operators in ‘use’/‘import’ limitation clauses and forwarding lists

```
use M only +, -;
import M.{<, >, <=, >=, !=, ==};
```

  - Note: ‘use M except *;’ was deprecated this release, so ‘*’ will only refer to the multiplication operator in the future

- Update ‘chpldoc’ and syntax highlighters to support ‘operator’ keyword

'MANAGE' STATEMENT

'MANAGE' STATEMENT
Background

- Python provides a language feature that can be used for *context management*

```
# Open a file, print it, and close it automatically at the end of the block.
with open('file.txt', 'r') as myFile:
  print(myFile.read())
```

- Context management is a feature with a lot of potential uses
  - E.g., managing files, timers, locks…
- Brought up when discussing ways to resize arrays of non-nilable classes
  - An array could be put into an unsafe state and resized within the scope of a managed block
- Decided to add context managers to Chapel due to these useful applications

'MANAGE' STATEMENT

This Effort

- Introduced the 'manage' statement to the Chapel language
  - Syntax and semantics are heavily inspired by Python

```
use IO;

// A type that emulates a file reader.
record reader {
  var path: string;
  var f: file;
}

// Assume the file given here always exists.
var rdr = new reader('file.txt');

// Use the reader to print the contents of the file.
manage rdr as c do write(c.lines());
```

```
proc reader.init(path: string) {
  this.path = path;
}

// Called when entering the block.
proc reader.enterThis() {
  f = open(path, mode=iomode.r);
  return f.reader();
}

// Called when leaving the block.
proc reader.leaveThis(in e: owned Error?) {
  f.close();
  if e then halt(e);
}
```

‘MANAGE’ STATEMENT

Status and Next Steps

- **Status**: The ‘manage’ statement is prototypical and is marked as ‘--warn-unstable’ in this release

- **Next Steps**: There are still a few design questions that need answers…
  - What should the enter/leave methods be named? Currently ‘enterThis()’ and ‘leaveThis()’

  - If multiple return intents exist for ‘enterThis()’, which overload should be called?

  - How should context managers interact with thrown errors besides guaranteeing that cleanup occurs?
    - E.g., in Python the ‘__exit__()’ method takes the thrown error as an argument

  - Should users be allowed to specify the storage of a resource? E.g…

```
// Here we specify that ‘count’ is a ‘var’.
manage m as var count do count += 1;
```

RESIZING NON-NILABLE ARRAYS

RESIZING NON-NILABLE ARRAYS

Background: Needed a general way to resize the domain of an array of non-nilable classes

- Decided the 'manage' statement could provide such a mechanism
  - The 'manage' statement is a new language feature added in this release

This Effort: A domain method that returns a context manager used to resize the domain

```
class C { var x = 0; }
var D = {0..0};
var A: [D] shared C = [new shared C(0)];

// Resize 'D', and manually initialize the new element of 'A'.
// A runtime error occurs if any element is uninitialized at the end of the block.
manage D.unsafeResize({0..1}, checks=true) do
  moveInitialize(A[1], new shared C(1));

writeln(A);  // Prints '{x = 0}, {x = 1}'
```

Status: Still being implemented; should be complete by the next release

RANGES / DOMAINS / ARRAYS: API CHANGES

RANGES / DOMAINS / ARRAYS: API CHANGES
Background

Background:
- As we work toward Chapel 2.0, we're reviewing methods/functions on built-in types
- In reviewing ranges/domains/arrays, we identified a few things that felt worth addressing in Chapel 1.25:

  – '.size' has traditionally returned 'idxType', which can be problematic

  ```
  …(-120..120:int(8)).size…                  // uh-oh, size exceeds max(int(8))
  …{1..20:uint(8), 1..20:uint(8)}.size…      // uh-oh, size exceeds max(uint(8))
  ```

  – 'array.indices' was nearly identical to 'array.domain'
    – returned a copy of the array's domain versus the domain itself
    – not particularly useful, and somewhat redundant

  – 'ident(r1: range(?), r2: range(?))' returned whether two ranges' defining (*low*, *high*, *stride*, *alignment*) values were identical
    – no other types support 'ident()', and it seems unlikely to be widely used

RANGES / DOMAINS / ARRAYS: API CHANGES

This Effort: '.size'

- Started making '.size' return 'int' for ranges/domains/arrays, as with other standard types and collections
  - rationale: 'int' is Chapel's default integer type and 'int(64)' is typically plenty large

- Added a '.sizeAs(type t: integral)' method to request the size to be returned as a particular integer type
  - rationale: for backwards-compatibility, or to query the size of a large collection as 'uint(64)'

  - to avoid breaking current code, added a warning when 'idxType != int'

```
…(-120..120:int(8)).size…
```

```
warning: 'range(int(8)).size' is changing to return 'int' values rather than 'int(8)'
  (to get the value as a different type, call the new method '.sizeAs(type t)')
  (to opt into the change now, re-compile with '-ssizeReturnsInt=true')
```

RANGES / DOMAINS / ARRAYS: API CHANGES

This Effort: '.indices'

- Decided to make '.indices' always return the domain's / array's indices as local values
  - e.g., for a Block-distributed array A, 'A.indices' now returns a non-distributed domain, like '{1..n, 1..n}'
  - rationale:
    - makes it more distinct from '.domain'
    - is often a useful thing to want to know, particularly when implementing distributions
    - other '.indices' queries on tuples, strings, lists, etc. also return a local representation
- As with '.size', used a warning to indicate this transition and a flag to opt-in early

```
…A.indices…
```

```
warning: the current behavior of  '.indices' on arrays is deprecated; see https://chapel-lang.org/docs/1.25/builtins/ChapelArray.html#ChapelArray.indices for details
```

- Open question: What should '.indices' do for sparse/associative arrays? [#18353]
  - in distributed cases, may not want to (or be able to) store all indices locally in a closed form
  - current proposal: '.indices' is an iterator for these cases, yielding indices locally

RANGES / DOMAINS / ARRAYS: API CHANGES

This Effort: ‘ident’

- Simply deprecated ‘ident()’ for now
  - rationale: strongly suspect that it’s completely unused
- Have proposed supporting it using ‘===’ if/when it is required [#17124]

RANGES / DOMAINS / ARRAYS: API CHANGES

Status and Next Steps

Status:

- Old behavior is preserved with warnings
- Users can opt into new behavior via compile-time flags
- Ready to finalize these in their new forms in the next release

Impact:

- For most user programs, only minimal changes should be required, if any
  - e.g., ‘idxType’ is most commonly ‘int’, so ‘.size’ won’t change in those cases
- These types are now closer to their expected Chapel 2.0 forms

Next Steps:

- Continue stabilizing the APIs for ranges, domains, and arrays, as well as other built-in types
- See the module review notes in “Ongoing Efforts” for details

SLICING ASSOCIATIVE ARRAYS

SLICING ASSOCIATIVE ARRAYS

Background and This Effort

Background:

- Chapel supports slicing arrays by domains

  …`MyArr[MyDom]`… *// refer to the sub-array of 'MyArr' defined by the indices in 'MyDom'*

- Traditionally, this support has focused primarily on rectangular arrays
  - e.g., slicing local or distributed dense arrays using dense or sparse domains

This Effort:

- Extended Chapel array slicing to support associative domains and arrays

```
const ProgModels = {"C", "Chapel", MPI", "OpenSHMEM", "UPC", "OpenMP"},
      Languages  = {"C", "Chapel", "UPC"};

var LinesOfCode: [ProgModels] int = …;

var langLines = + reduce LinesOfCode[Languages];    // sum the lines of code just for language-based solutions
```

SLICING ASSOCIATIVE ARRAYS
Impact and Next Steps

Impact:

- Permits interesting associative subarrays to be specified succinctly

Next Steps:

- Continue expanding slicing support to general mixes of array/domain types
  - relates to improving the definition of zippered iteration between regular and irregular arrays/domains

RANGES AND SINGLE-VALUE ENUMS

RANGES AND SINGLE-VALUE ENUMS

Background and This Effort

Background:

- The default value for a range is ‘1..0’ (or the closest equivalent using their ‘idxType’)
  - rationale: it’s an empty range, and it uses the two most basic values
- However, single-value enums don’t have multiple values, which leads to challenges:

```
enum color {black};  use color;
black..black           // OK: this is a range(color), no problem
var r: range(color);   // problem: what should r's value default to?  Recall, ranges represented as (low, high, stride, alignment)
```

- Traditionally, have not supported such single-value range types due to this challenge
- However, multiple users have requested such support over time

This Effort:

- Added support for such ranges
- Value effectively defaults to ‘0..<0’ as a special-case (e.g., ‘black..<black’ for ‘r’ above)
  - queries such as ‘.low’, ‘.size’ work as expected (e.g., returning ‘black’ and ‘0’ for ‘r’ above)
  - querying ‘.high’ returns ‘black’, but also prints an execution-time warning that the value can’t be trusted
- Of course, once the range is assigned ‘black..black’ (its only other legal value), everything works as expected

RANGES AND SINGLE-VALUE ENUMS
Status

Status:

- Single-value enum ranges are now supported
  - the main caveat being that '.high' is not a meaningful query for them
  - could be extended to other single-value types in the future
- The internal range module code was significantly cleaned up as a result of this effort
  - better initializers, error messages, etc.

Next Steps:

- Confirm that users are satisfied with this change

IMPROVEMENTS TO INTERFACES

INTERFACES OUTLINE

- Background
- This Effort
- Implementation Improvements
- More Design Questions
- Early Checking of Records and Mixed Generics
- Status
- Next Steps

INTERFACES
Background

- Interfaces for Chapel were proposed in CHIP 2
  - similar to *concepts* in C++20
  - provide cleaner specification and once-and-for-all type-checking of interface-constrained generics

- An initial implementation of interfaces has been available since 1.24 (see release notes)
  - enables experimentation with the feature
  - exposes a number of design questions — see issue #8629
  - uses hybrid resolution semantics
    - traditional generics are handled as before:
      - type-checked *late*, i.e., after instantiation
    - interface generics are:
      - type-checked *early*, i.e., before instantiation
      - then handled like traditional generics without late type-check

INTERFACES
This Effort

- Advanced the implementation of interfaces
  - enabled more use cases
  - revised some design decisions
  - exposed more design questions

- Investigated approaches for…
  - early checking of generic records
  - mixing interface-constrained and traditional generics

IMPLEMENTATION IMPROVEMENTS

INTERFACES

New Supported Use Cases 1/3

- ‘implements’ statements can now work with argument conversions and other call adjustments:

```
interface Drawable { proc draw(arg: Self): bool; }
int implements Drawable;                          // can now be implemented with any of the following:
proc draw(arg: real): bool { ... }                // using the coercion int → real
proc draw(arg: int, arg2: int = 0): bool { ... }  // using a default value
proc draw(arg: int) param: bool { ... }           // returning a ‘param’ (was not supported in 1.24)

interface Negatable { operator !(arg: Self): bool; }
implements Negatable(borrowed MyClass);                 // can now be implemented with this:
operator MyClass.!(arg: borrowed MyClass?): bool {...}  // using the coercion MyClass → MyClass?
                                                        // and adding an implicit receiver argument
```

- ‘implements’ statements can now work with promotion:

```
interface Drawable { proc draw(arg: Self): void; }
proc draw(arg: Box): void { ... }
implements Drawable([1..3] Box);   // uses a promoted version of draw()
```

INTERFACES

New Supported Use Cases 2/3

- An interface can now require a function that is itself an interface-constrained generic:

```
interface Drawable {
  proc draw(arg: Self, win: ?W) where W implements Window;
}

Box implements Drawable;
proc draw(arg: Box, win: ?W) where W implements Window   // an interface-constrained implementation
{ ... }
```

INTERFACES

New Supported Use Cases 3/3

- Given:

```
interface Drawable {
  type CT; CT implements Content;  // CT is an associated type
}
interface Content {
  proc bbox(arg: Self): 2*int;
}
```

- Can now pass an argument of associated type:

```
// can now pass 'content', whose type is an associated type, to another interface-constrained function
proc drawWithContent(shape: Drawable, content: arg1.CT) {
  helper(shape, content);
}
```

- Can now invoke a function available through an associated constraint:

```
// can now call 'bbox', which is available in the interface of the associated constraint 'CT implements Content'
proc helper(arg1: Drawable, arg2: arg1.CT) {
  ...bbox(arg2)...
}
```

INTERFACES

Revised Design Decisions 1/2

- Inference of implicit ‘implements’ statements is now off by default
  - switching to on-by-default in the future, if desired, will lead to fewer changes in user code
  - can be enabled using ‘--infer-implements-decls’

```
interface Drawable { proc draw(arg: Self): void; }
proc helper(arg: Drawable) { draw(arg); }
record Box { }
proc draw(arg: Box): void { ... }
helper(new Box());                    // requires --infer-implements-decls in 1.25
```

– With ‘--infer-implements-decls’, the compiler infers:

```
Box implements Drawable;
```

INTERFACES

Revised Design Decisions 2/2

- Implements statements with traditional generics are now checked late
  - traditional generics are ill-suited for early checking
  - the following code compiles in 1.25:

```
interface Drawable {
  proc draw(arg: Self): void;
}
proc helper(arg: Drawable) { draw(arg); }

record Box { var contents; }                // 'Box' is a traditional generic type

Box implements Drawable;                    // an error in 1.24

proc draw(arg: Box(string)) { ... }         // implements Drawable only for Box(string)

helper(new Box("hi"));    //  causes 'Box implements Drawable' to be checked for Box(string)
```

MORE DESIGN QUESTIONS

INTERFACES

More Design Questions 1/4

- How deep are required functions available through associated constraints?

```
interface Drawable_1 {  proc draw_1(arg: Self);
                        type AT1;  AT1 implements Drawable_2;
                        proc Self.get_1(): AT1;                   }
interface Drawable_2 {  proc draw_2(arg: Self);
                        type AT2;  AT2 implements Drawable_3;
                        proc Self.get_2(): AT2;                   }
interface Drawable_3 {  proc draw_3(arg: Self);
                        // ... and so on ...                      }

proc drawAll(arg1: Drawable_1) {  draw_1(arg1);
                                  draw_2(arg1.get_2());
                                  draw_3(arg1.get_2().get_3());
                                  // ... and so on ...            }
```

- Currently 3 levels are supported
  - an arbitrary choice intended to limit compilation time
- Should there continue to be a limit on interface nesting?
- For now, a helper function can be used when the limit is too restrictive

INTERFACES
More Design Questions 2/4

- Should an implementation of a required function preserve the names of the formals?
  - For example, Chapel's standard implementations of '==' currently use a variety of formal argument names
    - therefore, a user cannot require any particular names for '==' in their interfaces
  - This issue also comes up in overriding methods — see issue #11069

  - Need to be able to opt out of named-based argument passing to avoid the issue
    - especially when using third-party code to implement an interface

  - In 1.25 this check is not performed

INTERFACES
More Design Questions 3/4

- How can a function correlate the associated types of multiple interfaces?

```
interface HashtableEntry {
  type Key;
  type Val;
}
interface Hashtable {
  type Key;
  type Val;
}
// how to require 'table' and 'entry' to have matching 'Key' and 'Val' associated types?
proc addEntry(ref table: Hashtable, in entry: HashtableEntry) { ... }
```

  - Option 1: equality constraints

```
... where table.Key == entry.Key && table.Val == entry.Val ...
```

  - Option 2: explicit associated types

```
... in entry: HashtableEntry(Key = table.Key, Val = table.Val) ...
```

INTERFACES

More Design Questions 4/4

- How to support ‘param’ variations of an interface?
  - Many important cases in existing code, for example:
    - ‘RandomStream’ implementations support ‘param parSafe: bool’
    - an interface representing rectangular arrays will need to support ‘param dimension: int’
  - Option 1: stamp out an interface for all values of a ‘param’ in order to have complete early checking
    - not practical: would result in stamping out and early-checking a lot of unused code
  - Option 2: support *associated params* in interfaces

    ```
    interface Array { param rank; type eltType; ... }
    ```

    - treat params as non-param constants during “early” checking except when passing to param formals of required functions
    - post-process an interface-constrained generic upon instantiation to param-fold the conditionals etc.
- Post-processing will also enable other functionality (currently not supported), for example:
  - passing around types and params
  - compile-time checks “does this interface type implement that other interface?”

EARLY CHECKING OF RECORDS AND MIXED GENERICS

INTERFACES

Early Checking of Generic Records 1/3

- Generic records are currently treated as traditional generics
- Yet, early checking is desirable for methods just like it is for standalone functions
- Proposal: treat each generic field as an interface-constrained type when defining a method
  - For example, if a hash table implementation looks like this:

```
record hashtable {
  type keyType implements Hashable;
  type valType implements Copyable;
  var table: [] entry(keyType, valType);  // supposing array and 'entry' are traditional generic types
}
```

  - A 'fillSlot' method can be written like this:

```
proc hashtable.fillSlot(ref e: entry(keyType, valType), in k: keyType, in v: valType);
```

  - The compiler can interpret that method as if it were written like this:

```
proc (hashtable(?keyType, ?valType)).fillSlot(ref e: entry(keyType, valType),
                                                in k: keyType, in v: valType)
  where keyType implements Hashable && valType implements Copyable  { ... }
```

- Requires instantiating traditional generics (e.g., 'entry' above) with interface types during early checking

INTERFACES
Early Checking of Generic Records 2/3

- Instantiating traditional generics with interface types appears intuitive:

```
// given this type for a hash table entry:
record entry {
  var key;
  var val;
}
// … and a value instantiated with interface types keyType, valType, as on the previous slide
keyType implements Hashable; valType implements Copyable;
ref tableEntry: entry(keyType, valType)
// … then its fields have the respective interface types:
tableEntry.key : keyType
tableEntry.val : valType
```

  - availability of operations on such types is determined by the interfaces they implement

```
interface Hashable(Key) { proc hash(arg: Key): uint; }
…hash(tableEntry.key)… // this is OK, given that keyType implements Hashable
```

INTERFACES

Early Checking of Generic Records 3/3

- Chapel 1.25 supports an alternative approach to early checking of methods
  - avoids instantiating traditional generics with interface types
  - is based on introducing an interface that the generic record implements
  - is illustrated with a modification to modules/internal/ChapelHashtable.chpl:

```
// see test/constrained-generics/hashtable/MyHashtable.chpl
record hashtable { ... }
hashtable implements Hashtable;
interface Hashtable(HT) {          // implemented only by record hashtable
  type keyType; keyType implements Hashable;
  type valType; valType implements Copyable;
  type entryType;                  // represents the type entry(keyType, valType)
  type tableType;                  // represents the type array(entry(keyType, valType))
  proc HT.table ref: tableType;    // represents the field hashtable.table
}
```

  - hash table methods are defined as methods on this interface, e.g.:

```
proc Hashtable.fillSlot(ref e: this.entryType, in k: this.keyType, in v: this.valType) …
```

INTERFACES

Mixing Interface-Constrained and Traditional Generics

- Gradual conversion of existing generic code to interfaces is likely to lead to mixed generics, e.g.:
  - a function with interface-constrained and traditional generic formals
  - a record or class with interface-constrained and traditional generic fields
  - interface-constrained code that references traditional generic types or calls traditional generic functions

- Proposal: for a mixed generic, instantiate traditional generic types then treat it as interface-constrained

```
// if, in this declaration, 'keyType' is interface-constrained and 'valType' is traditional generic ...
proc (hashtable(?keyType, ?valType)).fillSlot
  (ref e: entry(keyType, valType), in k: keyType, in v: valType)
  where keyType implements Hashable { ... }
// ... check it only upon a concrete instantiation, e.g.. for this call where keyType=valType=int:
myTable.fillSlot(myEntry, 3, 5);
// ... retaining keyType as an interface-constrained generic and instantiating valType, as in:
proc (hashtable(?keyType,int)).fillSlot
  (ref e: entry(keyType, int), in k: keyType, in v: int)
  where keyType implements Hashable { ... }
```

STATUS AND NEXT STEPS

INTERFACES
Status

- The current implementation:
  - enables using interface-constrained generics in a variety of use cases
  - is ready for experimentation and learning about interfaces
  - reflects language design that is not yet stable
    - code using interfaces that works with 1.25 may not work in future releases as we refine the design

- 95 tests pass nightly testing, including early checking of methods:
  - a prototype conversion of modules/internal/ChapelHashtable.chpl
    - see test/constrained-generics/hashtable/MyHashtable.chpl
  - a partial conversion of modules/standard/Random.chpl
    - see test/constrained-generics/random/demo-random.chpl

INTERFACES
Next Steps

- Streamline the implementation
- Convert more Chapel libraries and internal code to interfaces
- Make progress on design discussions and decisions

OTHER LANGUAGE IMPROVEMENTS

OTHER LANGUAGE IMPROVEMENTS

For a more complete list of language changes and improvements in the 1.25 release, refer to the following sections in the CHANGES.md file:

- ‘Semantic Changes / Changes to Chapel Language’
- ‘New Features’
- ‘Feature Improvements’
- ‘Deprecated / Removed Language Features’

THANK YOU

https://chapel-lang.org
@ChapelLanguage