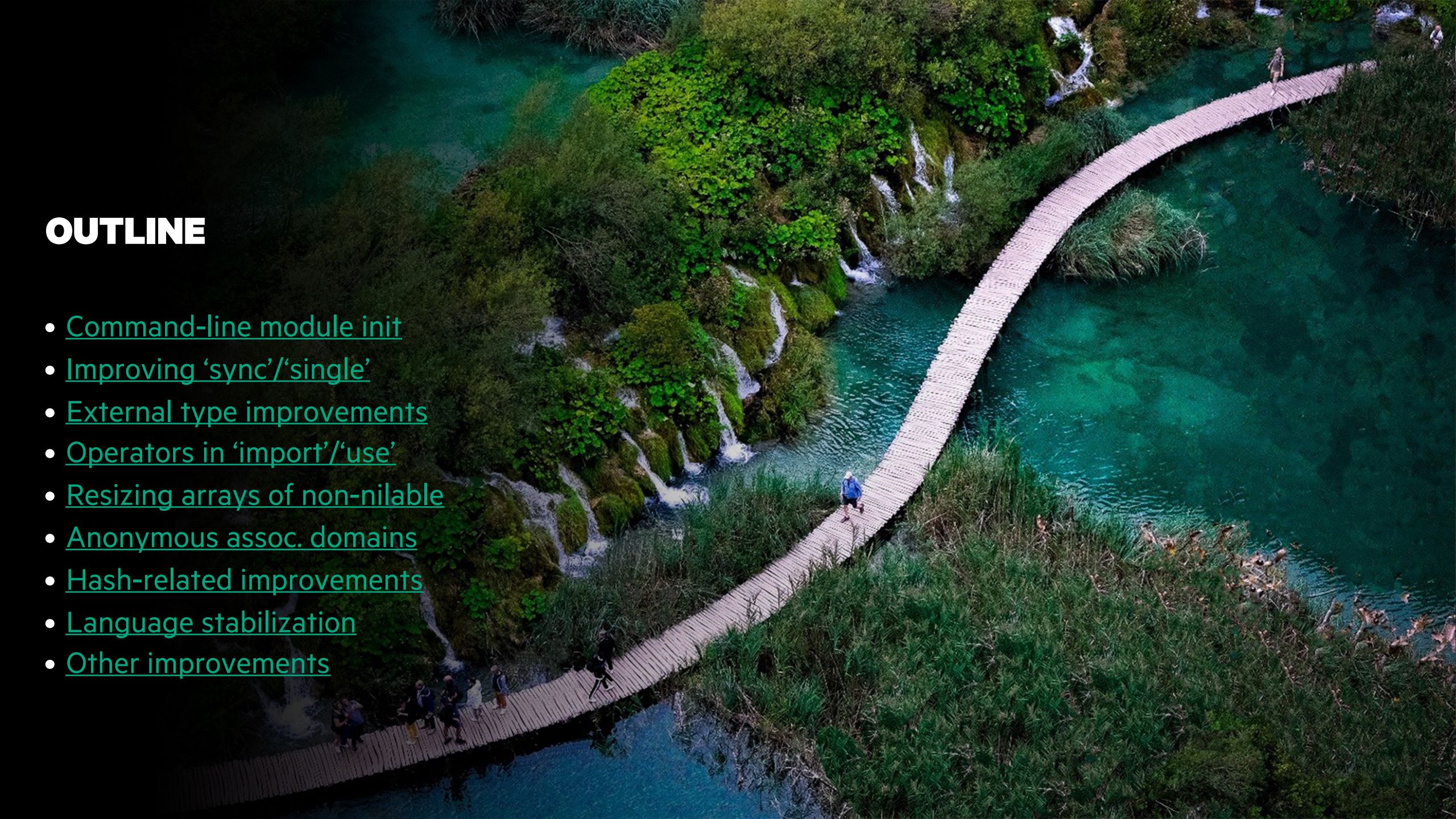
Hewlett Packard Enterprise

CHAPEL RELEASE NOTES, 1.25.1 / 1.26.0: LANGUAGE IMPROVEMENTS

Chapel Team

December 9, 2021 / March 31, 2022



OUTLINE

- Command-line module init
- Improving ‘sync’/‘single’
- External type improvements
- Operators in ‘import’/‘use’
- Resizing arrays of non-nilable
- Anonymous assoc. domains
- Hash-related improvements
- Language stabilization
- Other improvements

INITIALIZING MODULES FROM COMMAND-LINE FILES

INITIALIZING COMMAND-LINE MODULES

Background

- Module initialization consists of running the top-level statements within a module
  - For example, consider a ‘Hello World’ program:

```
module Hello {
  writeln("Hello World");    // this module-level statement runs when initializing the module ‘Hello’
}
```

- Historically, naming a module on the ‘chpl’ command line was insufficient to cause it to be initialized:

```
// a.chpl
proc main() {
  writeln("in a");
}
```

```
// b.chpl
module b {
  use a;
  writeln("in b");
}
```

```
$ chpl a.chpl b.chpl
$ ./a
in a
```

does not print “in b” because module ‘a’ (the main module) does not refer to it

- This behavior is surprising—intuitively, compiling ‘b.chpl’ should include it in the resulting program

INITIALIZING COMMAND-LINE MODULES

This Effort and Impact

This Effort: Adjusted the compiler to initialize top-level modules in files named on the command-line

- Improved the language specification to more clearly describe this and other facets of module initialization

Impact:

- Language design in this area is more intuitive
- Improved the behavior of the example program:

```
// a.chpl
proc main() {
  writeln("in a");
}
```

```
// b.chpl
module b {
  use a;
  writeln("in b");
}
```

```
$ chpl a.chpl b.chpl
$ ./a
in b
in a
```

- Also enables a more straightforward implementation of Arkouda modularization…

INITIALIZING COMMAND-LINE MODULES

Impact

- Consider a program like Arkouda, in which multiple modules register commands with a server module:

```
// Server.chpl
var server: …;

proc main() {
  var cmd: string;
  while readline(string) do
    server.process(cmd);
}
```

```
// M1.chpl
use Server;
const myCmds = …;
server.register(myCmds);
…
```

```
// M2.chpl
use Server;
const myCmds = …;
server.register(myCmds);
…
```

```
// M3.chpl
use Server;
const myCmds = …;
server.register(myCmds);
…
```

```
// M4.chpl
use Server;
const myCmds = …;
server.register(myCmds);
…
```

- With this change, distinct sets of modules can be trivially mixed together, leveraging self-registration

```
$ chpl Server.chpl M1.chpl M2.chpl M3.chpl M4.chpl  # build a server with all modules
$ chpl Server.chpl M2.chpl M3.chpl                  # build a server with just M2, M3
```

  - Previously, some piece of code would need to explicitly ‘use’/’import’/call each module for it to register itself

IMPROVEMENTS TO ‘SYNC’ AND ‘SINGLE’

SYNC AND SINGLE: BACKGROUND, THIS EFFORT, IMPACT

Background:

- 'sync' and 'single' are type modifiers that use full-empty semantics
  - e.g., 'mySync.readFE( )' means "read, blocking until the variable is full, leaving it empty"
- As of 1.25.0, they only supported these types:
  - 'nothing', 'bool', 'int', 'uint', 'real', 'imag', 'string'
  - enumerated types
  - 'unmanaged', 'borrowed', or 'shared' class types
- Notably, the following did not work with 'sync' or 'single':
  - 'owned' or non-nilable class types, user-defined record types, 'complex'

This Effort:

- Improved the generality of 'sync' and 'single' types to support all the types mentioned above
- For types that are not trivially copyable, 'readXX' on an empty sync now returns a default-initialized value
  - Enables the more common 'writeEF' 'readFE' pattern to move a value in and then out rather than copying
  - See example on the next slide

Impact:

- 'sync' and 'single' are significantly more capable as of 1.25.1

SYNC AND SINGLE: EXAMPLES

- Trivially copyable example:

```
var x: sync int;
x.writeEF(1);          // Sets 'x' to 'full' and stores '1' in it

var y = x.readFE();    // 'x' is now 'empty', and we read '1' out of it

var z = x.readXX();    // reads '1' since it was the last value stored, and because 'int' is trivially copyable
                       // (so, reusing the old value does not represent a memory error)
```

- Non-trivially copyable example

```
var a: sync string;
a.writeEF("hi");       // Sets 'a' to 'full' and stores "hi" in it

var b = a.readFE();    // 'a' is now 'empty' and we read "hi" out of it

var z = a.readXX();    // reads "" since 'a' was empty and 'string' is not trivially copyable
                       // (reusing the value also stored in 'b' might be a memory error if 'b' was deinitialized already)
```

ZERO-INITIALIZING VARIABLES OF EXTERN TYPE

ZERO-INITIALIZING EXTERNS

Background

- ‘extern’ types can refer to C types, for example:

```
// in C
typedef const void* syserr;
typedef struct { int64_t i; } mystruct_t;
```

```
// in Chapel
extern type syserr;
extern record mystruct_t { var i: int; }
```

- Historically, inner-scope variables of extern types were not initialized and had undefined values:

```
{
  var x: syserr;       writeln(x: int);   // causes segmentation fault
  var y: mystruct_t;   writeln(y.i);      // outputs an arbitrary number, e.g., 7
}
```

- This is surprising since Chapel variables are normally default initialized

```
{
  var z: int;          writeln(z);        // always outputs 0
}
```

- Note that module-level variables of extern types were already initialized to 0

ZERO-INITIALIZING EXTERNS

This Effort, Impact, and Status

This Effort: Variables of ‘extern’ type are now zero-initialized

- Next section describes how ‘init’ can be used to adjust default initialization for ‘extern’ records

Impact:

- Removed a source of bugs that has been coming up periodically for more than 5 years

Status:

- Included in 1.25.1

DEFINING INITIALIZERS FOR EXTERN RECORDS

EXTERN RECORD INIT
Background

- Historically, a default ‘init’ defined for an extern record had no effect:

```
// suppose a ‘mystruct_t’ is defined in C
extern record mystruct_t { var i: int; }

// users might expect this ‘init’ to be called for default initialization
proc mystruct_t.init() {
  writeln("in mystruct_t.init()");
  this.i = 1;
}

{
  var x: mystruct_t;   // does not print ”in mystruct_t.init()”
  writeln(x.i);        // used to output an arbitrary number, e.g,. ‘8’; with the previous change, would output ‘0’, but not ‘1’
}
```

- It is surprising that this program compiles, yet that the ‘proc init’ had no effect

EXTERN RECORD INIT

This Effort, Impact, and Status

This Effort:

- A ‘proc init’ defined for an extern record is now called for default initialization
- If no ‘proc init’ is provided, the extern record will be zero-initialized

Impact:

- Extern records are more flexible now
- A surprising behavior has been removed
- Extern and non-extern records are now more consistent

Status:

- Included in 1.25.1

RENAMING EXTERN TYPES

RENAMING EXTERN TYPES

Background and This Effort

Background:

- Sometimes an external identifier will have a name that is illegal, already in use, or unattractive in Chapel:

```
int type;                  // 'type' is reserved in Chapel, so we can't write 'extern var type: c_int;'
struct mystruct { … }      // the C name for this type is 'struct mystruct', but identifiers can't have spaces in Chapel
int read(…) { … }          // 'read' is already heavily overloaded in Chapel, so we may want to distinguish this case
typedef float imag;        // 'imag' is reserved in Chapel (and defined differently), so we can't write 'extern type imag = float;'
```

- Most 'extern' declaration forms in Chapel support the ability to give the external symbol name as a string:

```
extern "type" var c_type: c_int;                  // C name is 'type', but Chapel name is 'c_type'
extern "struct mystruct" record mystruct { … }    // C name is 'struct mystruct', but Chapel name is 'mystruct'
extern "read" proc c_read(…): c_int;              // C name is 'read', but Chapel name is c_read
```

  - However, extern 'type' declarations haven't supported this feature

This Effort:

- Add similar support for renaming in 'extern' type declarations

RENAMING EXTERN TYPES

Status and Impact

Status:

- External type declarations now support renaming as well
  - The following C declaration:

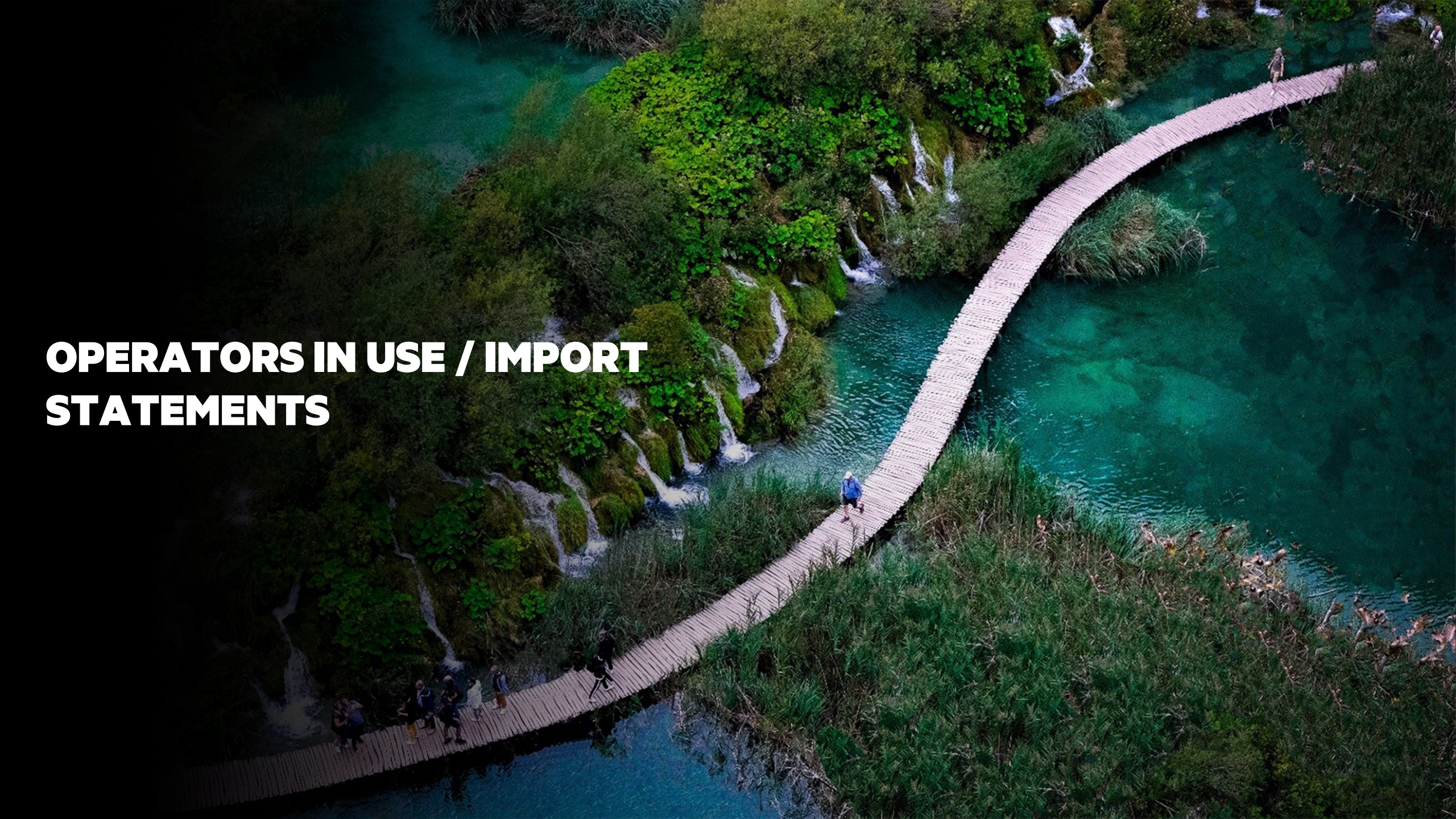
```
typedef float imag;    // 'imag' is reserved in Chapel, so we can't write 'extern type imag = float;'
```

  - Can now be written in Chapel as:

```
extern "imag" type c_imag = c_float;    // C name is 'imag', but Chapel name is 'c_imag'
```

Impact:

- Users can now rename external types for use in Chapel as needed / desired
- 'extern' type declarations are now more similar to other 'extern' declarations



OPERATORS IN USE / IMPORT STATEMENTS

OPERATORS IN USE / IMPORT

Background

- Directly controlling the visibility of operators via 'use' and 'import' statements was unsupported
  - Could potentially work around this in some cases, though imprecisely:

```
use Lib only +;              // syntax error
use Lib except a;            // work-around: would include '+', but also many other symbols

use Lib except -;            // syntax error
use Lib only a, b, c, d;     // work-around: excludes '–' but obnoxious to write if we want everything else in the module

import Lib.%;                // syntax error
```

  - No way to include some operators but not others

OPERATORS IN USE / IMPORT

This Effort and Next Steps

This Effort:

- Added support for listing operators in 'use' limitation clauses and 'import' statements

```
use Lib only +;      // now works!
use Lib except -;    // now works!
import Lib.%;        // now works!
```

Next Steps:

- Enable support for operators in forwarding clauses
  - Syntactically supported now, but has no effect

RESIZING ARRAYS OF NON-NILABLE CLASSES

RESIZING ARRAYS OF NON-NILABLE

Background and This Effort

Background: Resizing domains that govern arrays of non-nilable classes has triggered a halt

- Reason: No default value to use for any newly allocated elements

```
var D = {0..0};
var A: [D] shared C = [new shared C()];
D = {0..1};   // Halt: cannot resize domain
```

This Effort: Added an '.unsafeAssign' method to the domain

- Can be used to resize such domains and initialize any non-nilable array elements

```
var D = {0..0};
var A: [D] shared C = [new shared C()];
manage D.unsafeAssign({0..1}, checks=true) as mgr do
  for idx in mgr.newIndices() do                  // Loop over '1', the new index of 'D'
    mgr.initialize(A, idx, new shared C(idx));    // Initialize new element of 'A'
```

RESIZING ARRAYS OF NON-NILABLE

Impact, Status, and Next Steps

Impact: Can now use '.unsafeAssign' to resize domains governing arrays of non-nilable classes

- Use '.initialize' in the managed scope to manually initialize non-nilable array elements
- Manager provides '.isElementInitialized', and '.newIndices' as helper methods
- Optional runtime checking using 'checks=true' (defaults to 'false')

Status: The '.unsafeAssign' method can resize rectangular domains with arrays of non-nilable

Next Steps:

- Support resizing for arrays of all non-default-initializable types, not just non-nilable classes
  - requires a side data structure to track initializations since there's no obvious in-place sentinel value like for classes
- Associate default value of 'checks' to one of the compiler's '--no-*-checks ` flags
- Finalize behavior of arrays of default-initializable types within '.unsafeAssign'
- Support '.unsafeAssign' on associative domains and arrays
- Test lifetime checker support for the manager

CHANGES TO ARRAYS WITH ANONYMOUS ASSOCIATIVE DOMAINS

CHANGES TO ANONYMOUS ASSOCIATIVE DOMAINS

Background:

- Unlike most languages, Chapel supports distinct concepts for arrays and their index sets (domains)

```
const D = {1..10};          // represents the indices 1..10, inclusive
var A: [D] real;            // creates an array of 'real' values over D's indices
```

- For convenience/familiarity, a domain's curly brackets can be omitted for arrays over anonymous domains

```
var B: [1..10] real,        // no need to write 'var B: [{1..10}] real;'
    C: [1..3, 1..3] int;    // ditto for multi-dimensional arrays
```

- This convenience was also supported for associative arrays, which could be confusing:

```
var D: [1, 3, 7] real;      // is this a 3D array with a single element? No, it's an associative array, indexed by integers 1, 3, 7
```

- This form was not used often in practice and felt less well-motivated
  - since most languages don't support associative arrays, the familiarity argument from the rectangular case doesn't apply

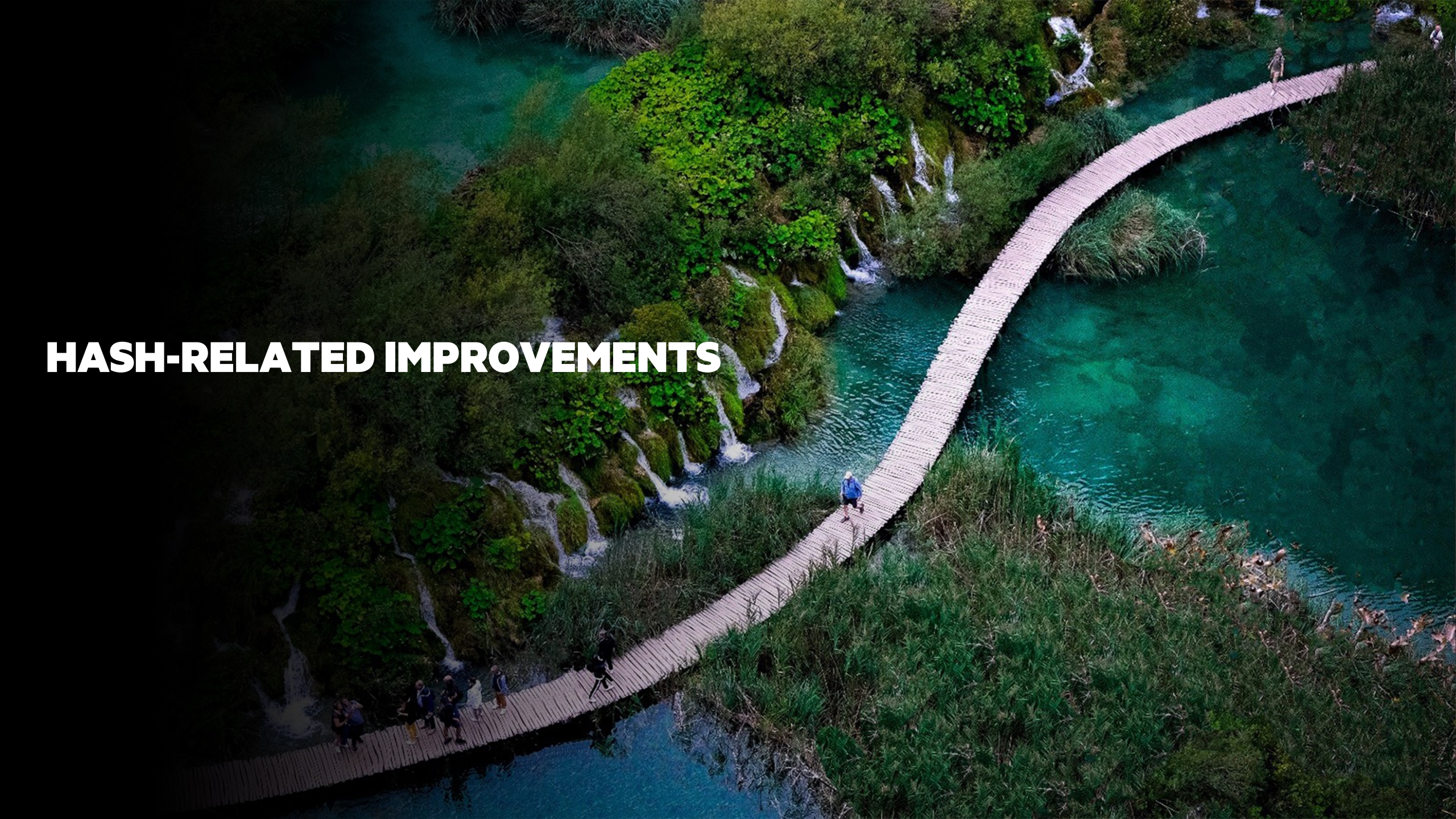
This Effort:

- Deprecated the ability to omit curly brackets for arrays over anonymous associative domains

Next Steps:

- See if users are concerned about this change, and remove support if not

HASH-RELATED IMPROVEMENTS



HASH IMPROVEMENTS

Background and This Effort

Background:

- Chapel supports several hash-based data structures:
  - associative domains and arrays in the language
  - the ‘set’ and ‘map’ collections in the standard library
- Users have requested better performance and flexibility for these types
  - including the ability to define their own hash functions
- Our k-nucleotide benchmark’s performance was much worse than other language implementations
  - a benchmark that looks at DNA sequences and calculates the frequency of certain patterns
  - overlap with user requests: wanted custom hash functions and better performance

This Effort:

- Made several changes to improve the flexibility, performance, and correctness of hash-related features

USER-DEFINED HASH FUNCTIONS

Background:

- Chapel previously generated hash functions for all records and classes with no ability to override the default
  - Prevented users from supplying a hash function for improved performance or when the default hash didn't work

This Effort:

- Added the ability for users to define a '.hash' method to override the default hash function
  - Called by the internal 'chpl__hashtable' type, used to implement Chapel's hash-based collections
  - Only supported on user-defined types—cannot override the 'int' hash method, for example

Impact:

- Added a '.hash' method to 'bigint', allowing its use with maps, associative domains, etc.
- 26% performance improvement to serial k-nucleotide benchmark:
  - Avoided a double hash that was otherwise required

Hash used	Execution time
Chapel-generated hash	29.31 s
User-defined hash	21.63 s

Next Steps:

- Finalize choice of '.hash' method name
  - Should we use a different name to avoid potential clashes with user identifiers? (see "Ongoing Efforts" release notes)

HASH TABLE IMPROVEMENTS

Background:

- Chapel's hash tables have traditionally used *quadratic probing* with a prime-number-sized table
- While investigating k-nucleotide, the prime-number-sized hash table was not performing well
  - Required an expensive modulus operator to find a slot in the hash table
  - Required resizing the hash table at half-capacity to guarantee finding an open slot

This Effort:

- Switched from prime-number-sized hash tables to using powers of 2 as the size
  - Supports replacing the modulus operator with a bitmask, which is a much cheaper operation
- Switched to *triangular probing*, which is guaranteed to find an open slot if one exists, regardless of table's size

Impact:

- 25% performance improvement to serial k-nucleotide benchmark:
- Allowed changes to the internal hash table resizing policies

Probing algorithm	Execution time
Prime-number probing	21.63 s
Triangular probing	15.13 s

HASH TABLE RESIZING POLICY IMPROVEMENTS

Background:

- 'chpl__hashtable' is the underlying data structure for Chapel's 'set', 'map', and associative domain/array types
- The quadratic probing algorithm we used requires that the table not exceed half capacity
  - The triangular probing algorithm we now use only requires a single open slot

This Effort:

- Added a 'resizeThreshold' to the hash table to control how full the table can be before resizing
  - e.g., a hash table of size 8 with a 'resizeThreshold' of 0.75 will resize when the 7^{th} element is inserted (it's > 75% full)
- Added an 'initialCapacity' to the hash table to set the starting size
  - Can avoid resizing altogether when the table size is known in advance
- These values also control how hash tables are reduced in size
  - Table shrinks by half when occupancy drops below 'resizeThreshold'/4
  - Table never shrinks below 'initialCapacity'
- Exposed both values through the initializers for 'set' and 'map'

LIMITING COMPILER-GENERATED HASH FUNCTIONS

Background:

- Traditionally, the compiler has generated a default hash function for every record
  - Approach was to hash each field, combining those hashes
- However, this may not always be appropriate
  - For example, imagine a record that represents a 'bigint' value using heap-allocated memory via a class or 'c_ptr'
  - Two records may represent the same 'bigint' value but using distinct classes/'c_ptr's
  - The compiler-generated hash would not work for these values since it can't know they represent the same value

This Effort:

- Decided that the compiler could not generate hashes for records that support a custom '==' or '!=' operator
  - Resolves cases like the 'bigint' example above since such types will need to support comparison operators to work
  - Seems appropriate given the use of these comparisons in resolving hash conflicts
- Squashed compiler-generated hash functions in these cases

Impact:

- Reduced the chances that the compiler will introduce a meaningless hash function

REFLECTING ABOUT HASHABLE TYPES

Background:

- Chapel's hash-based collections have traditionally issued errors for types "known" to be non-hashable
  - e.g., 'var DR: domain(range);' historically generated an error because ranges didn't support a hash function
  - Rationale: gave users a better error message than "could not resolve 'x.hash'" in an internal module
- However, this resulted in a maintenance issue
  - e.g., ranges have supported a hash function for years, yet Chapel 1.25.0 still generated the error message above

This Effort:

- Replaced list of unsupported types with 'Reflection' calls to determine a type's hashability

Impact:

- Enables hash-based types that should have been supported, yet were not
  - e.g., 'domain(range)' now works, as it should have for some time
- Reduces the burden on the development team to maintain the list of hashable types
- Meshes well with the previous slide, since records are no longer guaranteed to have hash functions

HASH IMPROVEMENTS

Status and Next Steps

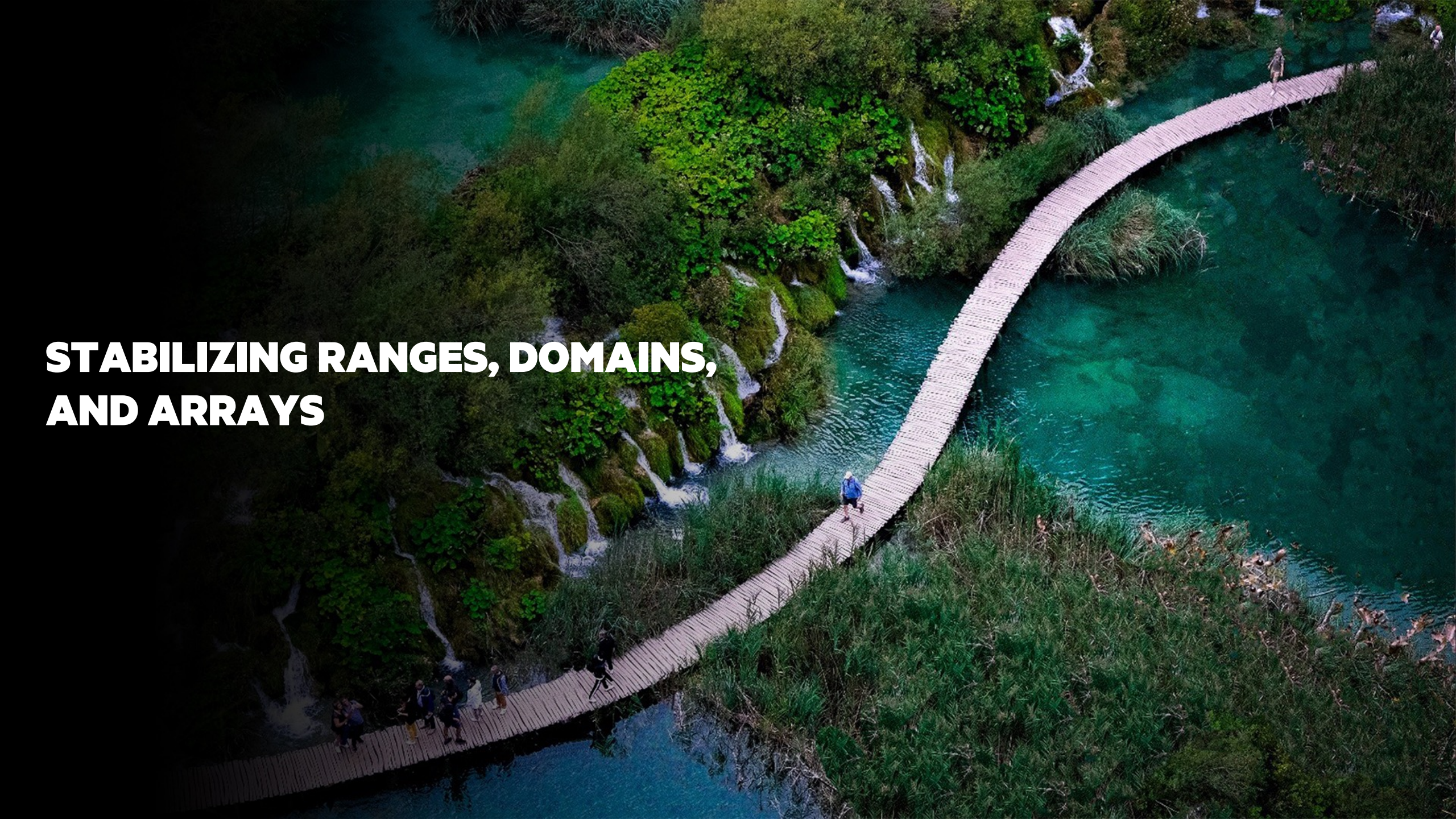
Status:

- Chapel's hash-based data structures are far more flexible, performant, and correct than they had been

Next Steps:

- Decide on a naming convention for special methods and use it for the 'hash' method

STABILIZING RANGES, DOMAINS, AND ARRAYS



STABILIZING RANGES, DOMAINS, AND ARRAYS

Background and This Effort

Background:

- We are reviewing features of ranges, domains, and arrays as part of the Chapel 2.0 stabilization effort
  - This is primarily happening as part of our standard library review, since these types are implemented in Chapel
- Review of these modules had started, but more work remained

This Effort:

- Continued working on improving methods and routines on ranges, domains, and arrays

STABILIZING RANGES

Background:

- In Chapel 1.25.0, we made some changes to the following features:
  - Deprecated support for '.size' and '.shape()' returning 'idxType', providing an opt-in to have them return 'int' instead
  - Deprecated 'range.ident' as a means of checking whether two ranges have identical (low, high, stride, alignment) tuples

Actions Taken:

- In Chapel 1.26.0:
  - '.size'/'.shape' now always return 'int'/'int' tuples for ranges, domains, and arrays
  - 'range.ident' is no longer available

Open Discussions:

- Potential changes to the range type itself:
  - Should we change the types/symbols used to characterize range types? [#17126, #17131]
  - Should the range type be generic, like 'domain'? [#18215]
- Should 'range.low', 'range.high' be aligned by default? [#17130]

STABILIZING DOMAINS

Actions Taken / Decisions Made:

- Replaced ‘domain.isSuper’ and ‘domain.isSubset’ with ‘domain.contains’
- ‘==’ and ‘!=’ between domains of different kinds, e.g., rectangular vs. associative, are now a compilation error
- Deprecated support for ‘|’, ‘&’, and ‘^’ on rectangular domains
- Improved error messages for unsupported operations on domains
- Changed standalone domain/array kind queries into methods
  - e.g., ‘isRectangularDom(MyDomain)’ -> ‘MyDomain.isRectangular()’
- Sparse domains and arrays will be considered unstable in Chapel 2.0

Open Discussions:

- Can we eliminate the runtime types for domains and arrays? [#19292]
- Should ‘+’ and ‘–’ mean translate (shift) on rectangular domains or set operations on irregular domains or …?
  - And should ‘|’, ‘&’, and ‘^’ mean set operations or promoted integer ops or compilation error or …? [#17101, #19254]
- Should (can?) ‘.hasSingleLocalSubdomain’ be ‘param’? [#11930]
- Better naming, terminology, and behavior of ‘dmapped’ keyword and ‘dist’ method [#17908]
- Is the difference between slicing with a domain vs. with a range too subtle? [#12936]

STABILIZING ARRAYS

Actions Taken:

- deprecated '.front()'/'.back()' on arrays, renaming them to '.first'/'.last'
- Changed behavior of '.indices' query to distinguish it from '.domain'
  - Rectangular arrays now return a local domain of indices; '.indices' on an irregular array is now a local iterator

Decisions made:

- Plan to move 'isArrayType', 'isArrayValue', 'isDmapType', 'isDmapValue' functions to 'Types' module
  - Since these are both auto-'use'd modules, from a user's perspective, this primarily affects documentation
- Plan to remove 'sorted', 'reverse', 'find', and 'count' methods from the array type
  - 'sorted' can be rewritten to copying to a temporary array and calling 'sort'
  - 'reverse' can be rewritten to a 'forall' loop without much difficulty
  - 'find' and 'count' can be written inline as a reduction

Open Discussions:

- Should 'idxType' return a tuple for multi-dimensional arrays? [#19141]
- Should we get rid of 'localSubdomain' in favor of 'localSubdomains' iterator [#19178]
- Should 'reshape' be made a method on arrays? Should we rename it? Should we mark it unstable? [#19176]

STABILIZING RANGES, DOMAINS, AND ARRAYS

Status and Next Steps

Status:

- Ranges, domains, and arrays continue to be more and more ready for Chapel 2.0

Next Steps:

- Continue to prioritize features on these types, given their centrality to the language

OTHER LANGUAGE STABILIZATION TOPICS

OTHER LANGUAGE STABILIZATION TOPICS

Background

Background:

- For the past several releases, we have generally been considering the core language ready for Chapel 2.0
  - as a result, most of our recent stabilization effort has focused on the standard libraries
- However, language stability questions still come up as a result of user/developer experiences
  - for example, the 'dyno' compiler revamp effort exposes issues as features are re-implemented

OTHER LANGUAGE STABILIZATION TOPICS

This Effort, Status, and Next Steps

This Effort:

- Wrestled with issues that seemed concerning with respect to language stabilization
  - shadowing / disambiguation, particularly w.r.t. 'public'/'private' 'use'/'import'
    - see issues #19306, #19167 , #19198, #19160, #19219, #19312, #19352, #19367, and the "Ongoing Efforts" deck
  - support for user-defined implicit conversions / array element accessor interface / ref intent overloading [#17999]
  - ergonomics, details of using classes [#19120, #19613, #19474]
- Also with issues that make the core language features for parallelism seem incomplete:
  - 'foreach' shadow variables, intents, and definition [#18500, #19153]
  - task-private variables for 'begin', 'cobegin', 'coforall' statements [#14659 , #15706]

Status:

- Discussions and implementation efforts are underway for many of these issues
- Level of attention varies depending on degree of severity

Next Steps:

- Resolve as many as possible prior to Chapel 2.0, prioritizing based on severity

OTHER LANGUAGE IMPROVEMENTS

OTHER LANGUAGE IMPROVEMENTS

For a more complete list of language changes and improvements in the 1.25.1 and 1.26.0 releases, refer to the following sections in the CHANGES.md file:

- ‘Syntactic / Naming Changes’
- ‘Semantic Changes / Changes to the Chapel Language’
- ‘New Features’
- ‘Feature Improvements’
- ‘Deprecated / Unstable / Removed Language Features’


THANK YOU
https://chapel-lang.org
@ChapelLanguage