Hewlett Packard Enterprise

CHAPEL 1.24 RELEASE NOTES: LANGUAGE AND LIBRARY IMPROVEMENTS

Chapel Team

March 18, 2021

CORE LANGUAGE STABILIZATION

CORE LANGUAGE STABILIZATION

Background and This Effort

Background:

- Over the past several releases, we have been working toward a forthcoming Chapel 2.0 release
- Intent: stop making backward-breaking changes to core language and library features
  - use semantic versioning to reflect if/when such changes are made

This Effort:

- Chapel 1.24 addresses most major language-related concerns identified in Chapel 1.23
  - resolution of tertiary methods and operators
  - various forms of conversions between types
  - definition of ‘out’ and ‘inout’ intents
  - nilability ergonomics w.r.t. conditional control flow
  - collections of non-nilable classes
  - implicit accesses to sync/single
- This deck addresses these efforts in detail

OUTLINE

USE AND IMPORT STATEMENT IMPROVEMENTS

USE/IMPORT IMPROVEMENTS
Background

- Could 'import' or 'use' types that were defined in a module
  - But couldn't explicitly bring in tertiary methods

```
import Mod.someMethod;  // error: can't list methods in limitation clauses
import Mod.R;           // error: 'R' not defined in 'Mod'
use Mod;                // This works, but isn't very precise
```

```
module Mod {
  // 'use' enables access to record 'R'
  private use DefinesR;
  proc R.someMethod() { … }
  proc returnAnR(): R { … }
}
```

- Needed a way to access methods when given an instance of a type
  - Methods were visible even if 'use' or 'import' was 'private' or had a limitation clause
    - This was inconsistent with other functions or symbols

```
import DefinesR, Mod.returnAnR;    // Nothing here explicitly brings in 'R.someMethod()'…

var rec = returnAnR();
rec.someMethod();                  // …but it could be called regardless
```

- Chapel 1.23 added support for always finding primary and secondary methods from a type's scope

USE/IMPORT IMPROVEMENTS
This Effort

- Enabled listing types with tertiary methods in 'use' / 'import'

```
use DefinesR;
import Mod.R;        // Now supported; provides access to 'proc R.*' in 'Mod'

var rec = new R(…);
rec.someMethod();   // Now works!
```

```
module Mod {
  // 'use' enables access to record 'R'
  private use DefinesR;
  proc R.someMethod() { … }
  proc returnAnR(): R { … }
}
```

- Returned to respecting 'private' and limitation clauses

```
import DefinesR, Mod.returnAnR;   // Nothing here explicitly brings in 'R.someMethod()'…

var rec = returnAnR();
rec.someMethod();                 // …so now it can't be called!
```

USE/IMPORT IMPROVEMENTS
Impact, and Next Steps

Impact:

- Privacy and limitation clauses are less complicated to explain
  - No more exceptions for methods

Next Steps:

- Fix bug with inherited type methods visibility (see issue #17134)

- Ensure operator methods have same visibility as regular methods

- Add support for listing operators in limitation clauses (see issue #17003)
  - E.g.

```
import Mod1.+;
use Mod2 except -;
```

OPERATOR OVERLOADING

OPERATOR OVERLOADING
Background

- Chapel permitted overloading operators via normal function definitions

```
proc +(lhs: t1, rhs: t2) { … }          // 't2' can be the same as 't1'
```

- Operators could not be defined as methods

- Could get an instance of a type without having its operators available

```
use DefinesR only R;   // Can't list '+' here!

var a = new R(5);
var b = new R(2);
var c = a + b;         // Error: the '+' operator isn't visible!
```

```
module DefinesR {
  record R {
    var f;
  }
  proc +(lhs: R, rhs: R)
    return new R(lhs.f + rhs.f);
}
```

- Made it difficult to associate operators with a type / find overloaded operators on a given type
  – Had to search for individual operators

OPERATOR OVERLOADING

This Effort

- Added a new ‘operator’ keyword to declare operator overloads

```
operator +(lhs: t1, rhs: t2) { … }    // ‘t2’ can be the same as ‘t1’
```

- Added support for declaring operator overloads as methods
  - Can be declared as primary, secondary, or tertiary methods
  - Treated like a ‘type’ method, but called as usual (infix/prefix/postfix notation, without a ‘this’ instance)

```
record R {
  var f: int;
  operator +(lhs: R, rhs: R) {    // Primary method
    return new R(lhs.f + rhs.f);
  }
}

operator R.-(lhs: R, rhs: R) {    // Secondary method
  return new R(lhs.f - rhs.f);
}
```

```
module DefinesTertiary {
  use DefinesR;

  // Tertiary method
  operator R.*(lhs: R, rhs: R) {
    return new R(lhs.f * rhs.f);
  }
}
```

OPERATOR OVERLOADING
Impact and Status

Impact:

- Operators can now be more closely associated with a type, like traditional methods
  - Can also still be declared as standalone functions for type-neutral cases
- The ‘operator’ keyword permits operators to be found more easily

Status:

- Standalone operators declared with the ‘operator’ keyword can be used in place of the ‘proc’ form
- Operator methods have some known issues (see next slide), but behave correctly in basic usage

OPERATOR OVERLOADING

Next Steps

- Replace ‘proc <op>’ definitions with ‘operator <op>’ definitions in internal/standard/package libraries
  - Use method approach where appropriate
- Deprecate ‘proc <op>’ form
- Decide how to handle forwarding operator methods (see issue #16992) and implement
- Add support for listing operators in forwarding and ‘use’/‘import’ limitation clauses (see issue #17003)
- Ensure method visibility rules apply to operator methods
- Update syntax highlighting modes to highlight ‘operator’ (emacs, vim, etc.)
- Make ‘operator’ visible in documentation of operator functions and methods declared with keyword
  - To make searching documentation easier as well
  - Today it looks like this for an operator method on type ‘Foo’:

```
proc type Foo.+(lhs: Foo, rhs: Foo)
```

LEVELS OF CONVERSIONS

LEVELS OF CONVERSIONS
Background

- Chapel has supported different kinds of conversions between types

```
proc f(in arg: real) { }
f(1);                             // implicit conversion for a function call

var x: real = 1.0;
x = 1;                            // conversion in assignment

var y: real = 1;                  // conversion in initialization

1: real;                          // cast
```

- There were open questions about the relationship among these conversions:
  - E.g., if an implicit conversion is allowed in initialization, should it also be allowed for function calls?

LEVELS OF CONVERSIONS

This Effort

- Developed a conceptual framework for these conversions and updated the language accordingly
  - Conversions between types come in 4 levels
  - Type authors should be able to choose any of these 4 levels
  - Distinguish between implicit conversions for function call arguments, assignment, and initialization
- So far, implicit call conversions only apply to built-in types
  - E.g., an ‘int’ argument passed to a function accepting a ‘real’
  - Allowing them for user-defined types should not impact the rest of these rules

Given	Which conversions are also required?		
	= (assign)	**init=** (initialize)	**:** (cast)
implicit call conversion	X	X	X
= (assign)		X	X
init= (initialize)			X
: (cast)			

LEVELS OF CONVERSIONS

This Effort

- Adjusted the compiler to generate an error when a required conversion is missing:
  - no ‘init=’ between two types when ‘=’ is present
  - no cast between two types when ‘init=’ is present

- Cleaned up some cases where the compiler translated a conversion into default-initialization + assign

- Added a user-facing way to define casts with ‘operator :’ as follows:

```
operator : (from: fromType, type t: toType) { ... }
```

LEVELS OF CONVERSIONS

Impact and Next Steps

Impact:

- Support for conversions is now expected to be stable
  - Even if user-defined implicit conversions for function calls are added later

Next Steps:

- Consider automatically generating ‘operator :’ from ‘init=’ when it is not provided
- Consider enabling implicit conversions for function calls
- Implement tertiary initializers and support them for all types
  - to allow conversions to be defined for tuples, arrays, integers, and other built-in types

‘OUT’ INTENT CHANGES

'OUT' INTENT CHANGES
Background

- We can think of the 'out' intent as creating a temporary variable at the call site and then assigning:

```
proc fOut(out arg: R) { ... }

fOut(c)
```

translates into

```
proc fOut'(arg: R) { ... }

var outTmp: R;
fOut'(outTmp);
c = outTmp;
```

  - Note that the 'out' intent uses '=' in some cases and 'init=' in others due to split init

- It was unclear what should happen when the 'out' formal has a different type from the actual argument:

```
proc int8Out(out arg: int(8)) { }
var myInt: int = 1;
int8Out(myInt);    // should this call resolve?  (traditionally, it hasn't)
```

- Similar code with 'return' was already working:

```
proc returnInt8(): int(8) { return 0; }
var myInt: int = 1;
myInt = returnInt8();    // works, uses a conversion when assigning to 'myInt'
```

‘OUT’ INTENT CHANGES

This Effort

- Changed ‘out’ intent to be more similar to ‘return’
  - the type of an ‘out’ intent formal is now inferred from the function body rather than the call site
  - types of ‘out’ intent formals are no longer considered in candidate selection or disambiguation
- Resolved open questions about how ‘out’ interacts with ‘=’ / ‘init=’ overloads
  - conversions enabled with ‘=’ / ‘init=’ can be run on a call to a function with ‘out’
  - these conversions still do not affect which function is called
- Adjusted ‘inout’ to reflect a composition of ‘in’ and ‘out’
  - the type is inferred from the call site, like ‘in’
  - conversions are considered in candidate selection, like ‘in’
  - as the called function is returning, actuals are set from the ‘inout’ formals with ‘=’ / ’init=’, like ‘out’
- Changed some ‘out’ intent arguments in modules and tests to ‘ref’
  - when the type information needed to flow from the call site

'OUT' INTENT CHANGES

Impact

- Conversions are now allowed for 'out' function calls using '=' / 'init=' as one might expect:

```
proc int8Out(out arg: int(8)) { }
var myInt: int = 1;
int8Out(myInt);    // now resolves, uses '=' to set 'myInt' from the 'out arg' formal
```

- Now 'out' formals can be used to initialize untyped variables

```
proc f(out a, out b, out c) {
  a = 1;
  b = 2.0;
  c = "hi";
}

var x, y, z;
f(x, y, z);

writeln( (x,y,z) );    // prints (1, 2.0, hi)
```

'OUT' INTENT CHANGES

Next Steps

- Allow programmers to request 'out' formal type inference from the call site (issue #17198)
  - 'channel.readbits' used to look like this

```
proc channel.readbits(out v: integral, nbits: integral): bool throws
```

  – however, that does not work if the type of 'v' is determined by the function body
  – for now, 'channel.readbits' uses the 'ref' intent:

```
proc channel.readbits(ref v: integral, nbits: integral): bool throws
```

  - a type query expression could indicate that the type should come from the call site:

```
proc channel.readbits(out v: ?T, nbits: integral): bool throws
```

  - such a mechanism could enable a few other patterns as well:

```
proc foo(out B: ?T) where isArray(T) { for i in B.domain do B(i) = i; }
proc f(out arg: ?T) { if something then arg = 1; }
```

CONTROL-FLOW DECLARATIONS

CONTROL-FLOW DECLARATIONS

Background and This Effort

Background: a nil-check is required yet unnecessary after establishing that a nilable variable is non-nil

```
var c: MyClass? = ...;
if c then
  c.doSomething();    // error: did you mean 'c!.doSomething()' ?
```

This Effort: the non-nilable value can be stored in a “control-flow variable” or “constant”

```
if const c2 = c then
  c2.doSomething();          // OK: 'c2' is non-nilable
```

- also available in while-do loops

```
while const curr = computeNext() do
  curr.process();            // OK: 'curr' is non-nilable
```

- a control-flow variable is accessible only in the corresponding then-branch or loop body
- if it is declared as ‘var’, it can be assigned
- a control-flow variable stores a ‘borrow’ when its control-flow expression is ‘owned’ or ‘shared’

CONTROL-FLOW DECLARATIONS
Impact and Next Steps

Impact:

- improved nilability ergonomics
- superfluous postfix-! operations can now be avoided

Next Steps:

- potentially permit the control-flow variable to retain ‘owned’ or ‘shared’ management when desired
- potentially consider allowing other types in the control-flow declarations, like numbers

REORGANIZING THE MEMORY MODULE

REORGANIZING THE MEMORY MODULE
Background and This Effort

Background: The ‘Memory’ module contained functions to diagnose memory usage

- E.g., ‘memUsed()’ returned the memory usage of the current locale
- E.g., ‘physicalMemory()’ returned the total memory on a locale

This Effort: Expanded the capabilities of the ‘Memory’ module

- Reorganized the ‘Memory’ module into submodules
  - ‘Memory’: The root module
    - ‘Memory.Diagnostics’: The contents of the ‘Memory’ module were moved here
    - ‘Memory.Initialization’: New functions for low-level moves and deinits
- Deprecated the functions and types in the ‘Memory’ module

REORGANIZING THE MEMORY MODULE

The ‘Memory.Initialization’ Module

- The ‘Memory.Initialization’ module provides functions to perform low-level moves

  - A low-level move copies the bytes of a value around in memory
    - Like C assignment or C ‘memcpy()’

  - A low-level move does *not* perform assignment (e.g., Chapel’s proc=)
    - The destination is overwritten, and leaks/crashes can occur if used improperly

  - A low-level move does *not* produce a copy (e.g., Chapel’s init=)
    - The ‘moveInitialize()’ function will error if calling it would copy ‘src’

```
var src: nonPodRecord;
var dst: nonPodRecord = noinit;  // Assume ‘noinit’ works for types besides arrays (it currently does not)
moveInitialize(dst, src);        // Compiler error: Call to ‘moveInitialize’ would copy ‘src’
writeln(src);
```

REORGANIZING THE MEMORY MODULE

The 'Memory.Initialization' Module

- The 'Memory.Initialization' module provides tools to help users build their own collections

```
use Memory.Initialization;

record myList {
  // Assume 'noinit' works for non-POD types (right now it does not)
  var data: [0..7] nonPodRecord = noinit;
  var size = 0;
}
proc myList.add(in x: nonPodRecord) {
  // Move 'x' into 'data' without assigning it
  moveInitialize(data[size], x);
  size += 1;
}
proc myList.popLast() {
  size -= 1;
  return moveToValue(data[size]); // Consume data[size] and move it into a new value
}
```

```
proc myList.clear() {
  for i in 0..<size {
    // Destroy elements of a 'myList' manually
    explicitDeinit(data[i]);
  }
}
```

REORGANIZING THE MEMORY MODULE
Impact, Next Steps

Impact: The ‘Memory’ namespace has been expanded for use by several memory-themed modules
- Users should change uses of ‘Memory’ to ‘Memory.Diagnostics’ to avoid deprecation warnings

Next Steps:
- Make the compiler aware of calls to ‘explicitDeinit()’
  - It still deinitializes variables that have had ‘explicitDeinit()’ called on them
- Update standard collections to use ‘Memory.Initialization’ where possible
  - Strive to build our collections entirely out of user-facing features
- Explore more kinds of low-level moves
  - Such as a function to move a value across locales, see: #15808

SUPPORT FOR ARRAYS OF CLASSES

SUPPORT FOR ARRAYS OF CLASSES
Background

Background: Only one flavor of fixed size array was left unsupported in the previous release

	list	map	set	fixed array	resized array	assoc array	sparse	tuple
owned t	✅	✅	✅	✅	🔷	🔷	🔷	✅
shared t	✅	✅	✅	✅	🔷	🔷	🔷	✅
borrowed t	✅	✅	✅	✅	🔷	🔷	🔷	✅
unmanaged t	✅	✅	✅	✅	🔷	🔷	🔷	✅
(shared t, shared t)	✅	✅	✅	❌	🔷	🔷	🔷	✅
owned t?	✅	✅	✅	✅	✅	✅	✅	✅
shared t?	✅	✅	✅	✅	✅	✅	✅	✅
borrowed t?	✅	✅	✅	✅	✅	✅	✅	✅
unmanaged t?	✅	✅	✅	✅	✅	✅	✅	✅
(shared t?, shared t?)	✅	✅	✅	✅	✅	✅	✅	✅
record	✅	✅	✅	✅	✅	✅	✅	✅

Key

✅ Working ❌ Not yet working 🔷 Not expected to work

SUPPORT FOR ARRAYS OF CLASSES

This Effort

This Effort: Added support for fixed arrays of tuples containing non-nilable classes

	list	map	set	fixed array	resized array	assoc array	sparse	tuple
owned t	✅	✅	✅	✅	🔷	🔷	🔷	✅
shared t	✅	✅	✅	✅	🔷	🔷	🔷	✅
borrowed t	✅	✅	✅	✅	🔷	🔷	🔷	✅
unmanaged t	✅	✅	✅	✅	🔷	🔷	🔷	✅
(shared t, shared t)	✅	✅	✅	✅	🔷	🔷	🔷	✅
owned t?	✅	✅	✅	✅	✅	✅	✅	✅
shared t?	✅	✅	✅	✅	✅	✅	✅	✅
borrowed t?	✅	✅	✅	✅	✅	✅	✅	✅
unmanaged t?	✅	✅	✅	✅	✅	✅	✅	✅
(shared t?, shared t?)	✅	✅	✅	✅	✅	✅	✅	✅
record	✅	✅						✅

✅ Working

- Bug related to default initialization of tuple array elements containing non-nilable classes
- Fixed by #16802

SUPPORT FOR ARRAYS OF CLASSES
Impact and Next Steps

Impact: Fixed-size arrays of all class flavors are now supported

Next Steps:

- Support resized arrays of non-nilable classes
  - see discussion in ‘Ongoing Efforts’ release notes

DEPRECATING IMPLICIT READS/WRITES OF SYNC/SINGLE

IMPLICIT SYNC READS/WRITES

Background

- Since Chapel's inception, sync/single variables have supported implicit accesses:

```
var count$: sync int;
count$ = count$ + 1;    // equivalent to the more explicit: 'count$.writeEF(count$.readFE() + 1);'
```

- Rationale:
  - more convenient than requiring methods for every read/write
  - followed the precedent set by the Tera MTA / Cray XMT programming model
- However, this has also been a source of long-term concern:
  - unwitting reads/writes to such variables can cause deadlocks
    - this led to the convention of naming sync/single variables with a '$' to alert programmers to their presence
    - yet, it's arguably a red flag for a language to depend on a naming convention to ensure clarity
  - has also resulted in some asymmetries in the language, e.g.:

```
var x = y;          // in most cases, x.type == y.type
var z = count$;     // but here, z.type == int
```

- This stood out as a core language feature we'd likely regret freezing as-is

IMPLICIT SYNC READS/WRITES

This Effort

- Updated Chapel's modules and tests to use explicit read/write methods
- Deprecated implicit reads/writes of syncs/singles
  - given:

```
var s$, s2$: sync int;
```

  - the following patterns now generate warnings about implicit reads/writes being deprecated:

```
var x = s$;                      // rewrite:  var x = s$.readFE();
s$ = 1;                          // rewrite:  s$.writeEF(1);
s$ += 1;                         // rewrite:  s$.writeEF(s$.readFE() + 1);
s$ = s2$;                        // rewrite:  s$.writeEF(s2$.readFE());
… s$ + s2$ …;                    // rewrite:  … s$.readFE() + s2$.readFE() …
f(s$);  proc f(x: int) {…}       // rewrite:  f(s$.readFE())
if s$ then …                     // rewrite:  if s$.readFE() then …
```

  - warnings are of the form:
    - warning: Initializing a type-inferred variable from a 'sync' is deprecated; apply a '.read??()' method to the right-hand side
    - warning: Direct assignment to 'sync' variables is deprecated; apply a 'write??()' method to modify one
    - etc.

IMPLICIT SYNC READS/WRITES

Impact and Next Steps

Impact:

- New warnings should encourage users to stop relying on implicit reads/writes so that we can remove them

Next Steps:

- Determine how compiler-generated initializers of objects with sync/single fields should work
  - see next slide
- Remove support for implicit reads/writes
- Consider ceasing to recommend that sync/single variables be decorated with '$'
- Implement default I/O for syncs/singles
  - the following has traditionally not been supported due to questions about whether to interpret it as 'writeln(s$.readFE());'

    ```
    writeln(s$);
    ```

  - but without implicit reads/writes, it seems more obvious to treat it as IO on the sync/single itself
    - e.g., perhaps write the value if full, a string like '<empty>' if not?

IMPLICIT SYNC READS/WRITES

Next Steps: Compiler-generated Initializers

- Given:

```
class C {
  var s: sync int;
}
```

- Traditionally, the compiler has generated:

```
proc C.init(s: sync int) {
  this.s = s;    // this generates a warning today, requiring the user to specify an initializer if they want to avoid it
}
```

- More useful would be to have the compiler generate:

```
proc C.init(s: int) {
  this.s = s;
}
```

  - rationale: the only ‘init=’ routine that a sync variable supports has the form:

```
proc (sync t).init=(rhs: t) { … }
```

  - open question: would such behavior be specific to syncs/singles, or applicable to user types as well?

CORE LANGUAGE STABILIZATION: NEXT STEPS

CORE LANGUAGE STABILIZATION
Next Steps

- Knock out remaining language issues:
  - Complete operator methods, deprecating ‘proc’-style operator overloads
  - Resolve sync field initializer issues and remove implicit sync accesses
  - Complete work on arrays and collections of non-nilable classes
- Focus increasingly on standard library stabilization
- Complete interfaces
- Continue to explore impact of:
  - capture of iterator expressions into untyped variables:

    ```
    var x = [i in 1..10] i;    // what is the type of ‘x’?
    ```
  - 0-tuples

CORE LANGUAGE STABILIZATION

Next Steps: Longer-term

- After Chapel 2.0, what else remains to be stabilized / defined?
  - first-class functions
  - ability to create records with non-default behaviors (e.g., argument / task intents)
  - interoperability features
  - partial reductions, scans
  - how parallel and zippered iterators are defined
  - user-defined reductions and scans
  - user-defined domain maps
  - ability to disable pass-by-keyword matching
  - …

OTHER LANGUAGE AND LIBRARY IMPROVEMENTS

OTHER LANGUAGE AND LIBRARY IMPROVEMENTS

For a more complete list of language and library changes and improvements in the 1.24 release, refer to the following sections in the CHANGES.md file:

- ‘Syntactic / Naming Changes’
- ‘Semantic Changes / Changes to Chapel Language’
- ‘Namespace Changes’
- ‘New Features’
- ‘Feature Improvements’
- ‘Deprecated / Unstable / Removed Language Features’
- ‘Deprecated / Removed Library Features’
- ‘Standard Library Modules’
- ‘Package Modules’
- ‘Standard Domain Maps (Layouts and Distributions)’

THANK YOU

https://chapel-lang.org
@ChapelLanguage