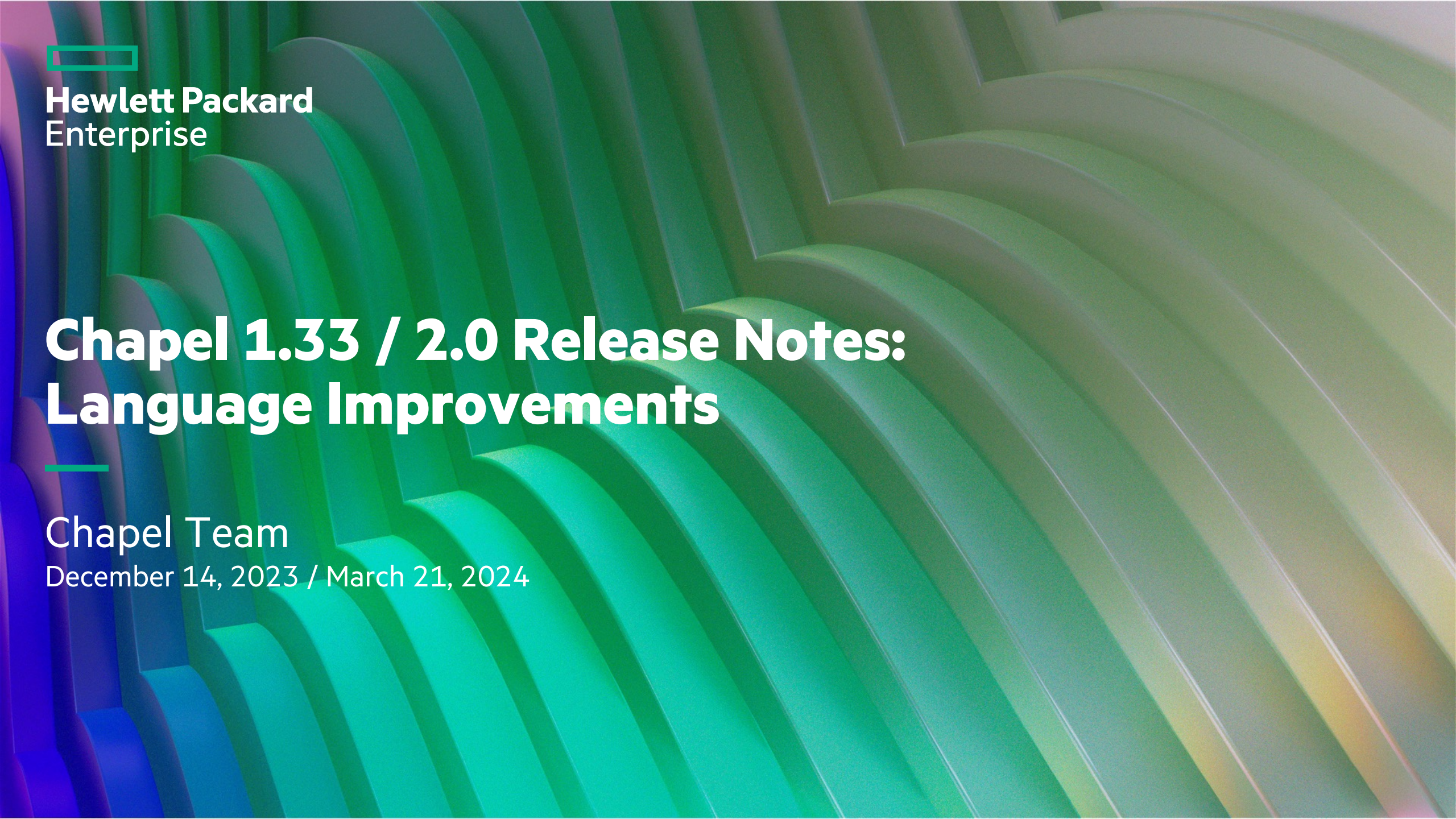


Chapel 1.33 / 2.0 Release Notes: Language Improvements

Chapel Team

December 14, 2023 / March 21, 2024

Outline

- Chapel 2.0 Stabilization
- Array Default Task Intents
- “Static” Variables
- Appending to ‘bytes’/‘string’

Chapel 2.0 Stabilization

Chapel 2.0 Stabilization
Background

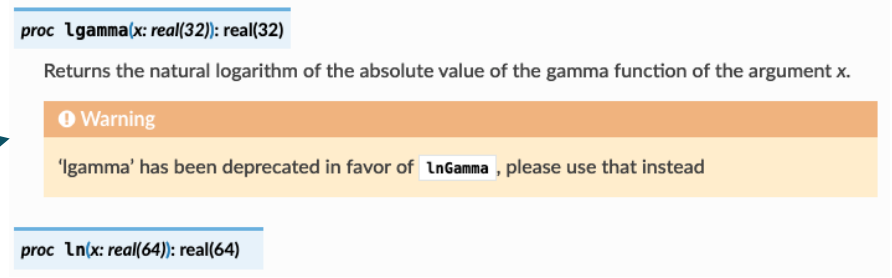
- Have been driving towards 2.0 milestone
  - Core language and library features are considered stable
    - Any future changes made to these features will be backward-compatible
    - (Or, would trigger bumping the version number to Chapel 3.0)
  - Some features are noted as unstable in the documentation and trigger warnings when using '--warn-unstable'

- Chapel 1.32 was the first 2.0 Release Candidate
  - 35 modules stabilized
  - Various adjustments to core language features
  - Users were encouraged to give feedback if anything needed tweaking
    - Any changes made based on this feedback would get incorporated into the official 2.0 release

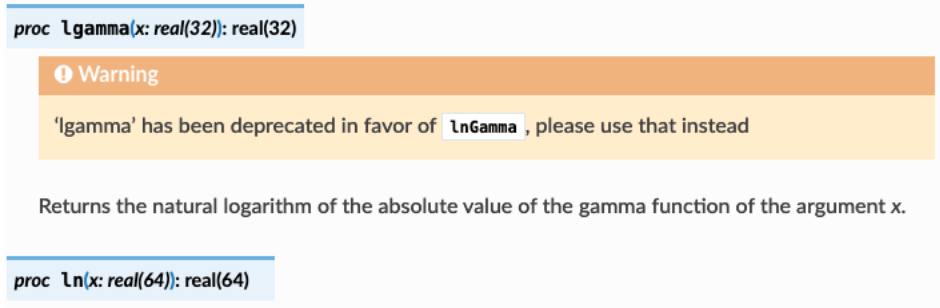
Chapel 2.0 Stabilization
This Effort

- The Spring 2024 release is the official 2.0 release! (see announcement post on blog)
  - Additional features were stabilized since 1.32 (see overview post on blog)
    - 'Random' module
    - Default task intent for arrays
    - Casting to 'unmanaged'
    - Associative domains when 'parSafe=false'
  - Adjustments were made to the 'IO' module
    - to the binary format (based on user feedback)
    - to the locking behavior of 'fileReader' and 'fileWriter'
  - Documentation was improved
    - Added missing documentation to stable modules
    - Adjusted the placement of deprecation/unstable warnings
      - Now more obviously associated with the impacted symbol

Before:



After:



Chapel 2.0 Stabilization
This Effort

- Additional warnings/errors were added
  - Comparison operators ('>', '<=', etc.) are no longer chainable without parentheses

```
if a < b < c then …   // was allowed, now a syntax error
```

  - Indirectly modified arguments that were inferred to be 'const ref' will now generate an unstable warning

```
var globalRec = new myRecord(15);
foo(globalRec);
proc foo(const r: myRecord) {
  globalRec.x = 3;   // indirectly modifies 'r'
}
```

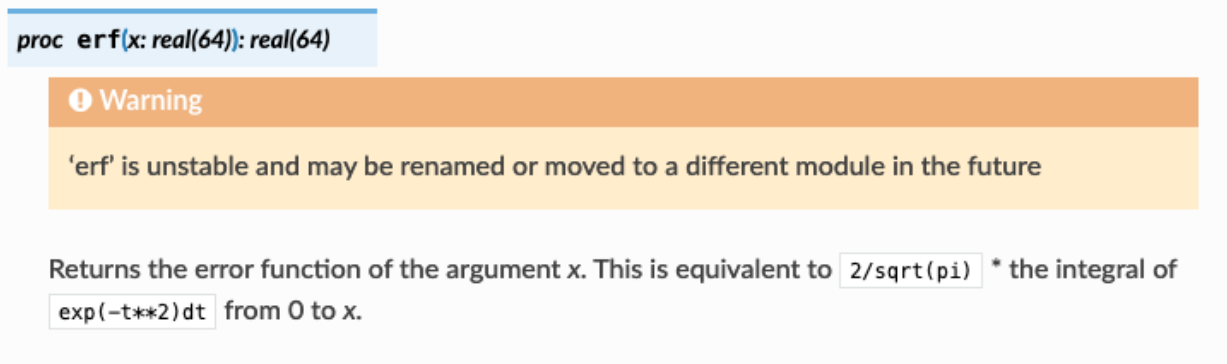
- Fixed a bug preventing some deprecation/unstable warnings from firing
  - You may see additional warnings for 'owned' and 'shared' because of this fix, e.g.

```
var o: owned MyClass? = new owned MyClass(4);
var s: shared = o;   // now properly generates "warning: assigning owned class to shared class is deprecated."
```

Chapel 2.0 Stabilization
Impact

- Programs that use only stable features shouldn't require updates in future releases
  - And it's easier than ever to write such programs

- It's also easier to determine which features are stable and which are unstable
  - Via documentation:



  - Via compiling with '--warn-unstable':

```
$ chpl --warn-unstable callErf.chpl
callErf.chpl:3: warning: 'erf' is unstable and may be renamed or moved to a different
module in the future
```

Chapel 2.0 Stabilization

Next Steps

- Continue to respond to user feedback about what to stabilize next
- Ensure new features get reviewed with stabilization in mind
  - To reduce the need for future changes
- Continue stabilizing unstable features
  - 'foreach'
  - 'dmapped' keyword
  - 'Sort' module
- Create a process for reviewing changes going forward (breaking or non-)
  - E.g., new keywords, “obviously” broken features
  - Create a board of users to guide the language’s evolution?
- Consider generating unstable warnings by default as more features are stabilized

Default Task Intents for Arrays

Default Task Intents for Arrays
Background

- In 1.32, the default array argument and task intent changed
  - default array argument intent became ‘const’
  - default array task intent for other parallel constructs (i.e. ‘coforall’, ‘begin’, and ‘cobegin’) became ‘const’
  - default array task intent for ‘forall’ loops remained ref-if-modified

```
proc myFunc(ref A:[], B:[]) do
  A = B;

begin with (ref A)
  A = B;

forall i in A.domain /* with (ref A) */ do
  A[i] = i;
```

- It was unfortunate that default array task intents on ‘forall’ loops differed
  - yet, users also considered unifying on ‘const’ to be too intrusive on common code idioms

Default Task Intents for Arrays

This Effort

- Default array intents for all parallel constructs are now unified to a new approach:
  - the default intent is now inferred from the outer variable
  - i.e., if the array is modifiable outside the loop, it is modifiable inside the loop

```
var   A: [1..10] int;
const B: [1..10] int;

[i in 1..10] A[i] = i;   // ref intent for A inferred from 'var A' variable

forall i in 1..10 with (ref B) do
  B[i] = i;              // error: cannot assign to const variable

proc myFunc(ref A:[], const B:[]) {
  forall i in B.domain do
    B[i] = i;            // error: cannot assign to const variable
  begin                  // ref intent for A inferred from 'ref A' formal argument
    A = B;
}
```

Default Task Intents for Arrays
Impact

- Reinstated promoted array indexing, adhering to the new default intent rule
  - previously, this feature had relied upon ref-if-modified
  - to alleviate concerns that this code pattern was unsafe, added '--warn-potential-races'

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

A[B] += 1;
```

```
> chpl main.chpl --warn-potential-races
main.chpl:4: warning: modifying the result of a promoted index expression is a potential
race condition
```

- New rules seem to achieve the right mix of convenience and safety
  - user idioms remain cleaner, as requested
  - consistency has been reinstated across parallel constructs
  - safety is encouraged via the 'const' default argument intent and its propagation into parallel contexts

Static Variables

Static Variables

Background and This Effort

Background:

- C and C++ have variables that persist across invocations of a function

```
void f() {
  static int x = 10;
}
```

- Can be used for mutable data (e.g., counters) between invocations
- Or, to avoid re-running expensive computations multiple times (e.g., computing lookup tables)

This Effort:

- Added prototype support for static local variables to Chapel

```
proc f() {
  @functionStatic
  var x = 10;
}
```

Static Variables
Impact

Impact: Supports caching computations local to a routine

- particularly useful within generic routines, where a module-scope variable can't be used as a workaround

```
proc computeExpensiveFibonacciTable(param tableSize: int, type t): c_array(t, tableSize) {
  writeln("Computing expensive table");
  // computes and returns a C array of Fibonacci numbers, represented as type 't'...
}

proc getNthElement(x: int, type t=int): t {
  @functionStatic
  const table = computeExpensiveFibonacciTable(94, t);
  return table[x];
}

writeln(getNthElement(0));   // prints 'Computing expensive table' then '1'
writeln(getNthElement(1));   // prints '1'
writeln(getNthElement(2));   // prints '2'
writeln(getNthElement(3));   // prints '3'
writeln(getNthElement(4));   // prints '5'
```

Static Variables
Status, and Next Steps

Status:

- An initial prototype is in the Chapel 2.0 release, but is unstable
- Static variables are synchronized using Chapel's atomic types out of the box
- Static variables support multi-locale execution (stored on first locale to initialize the variable)
- There are some limitations:
  - variable must be initialized directly and only once (i.e., no split initialization, default initialization)
  - arrays and domains not supported due to their runtime type information (example uses 'c_array')
  - no support for replication across locales, yet

Next Steps:

- Investigate support for replication strategies (e.g., precomputed value is replicated across all locales)
- Investigate proper language-level support (e.g., keyword rather than attribute)
- Investigate support for static variables with runtime types
- Optimize implementation for 'var'/'const' declarations


Appending Numeric Values to
bytes/string

Appending to 'bytes' / 'string'
Background

- Historically, appending a numeric byte value to a 'bytes' has been awkward and slow
- Comes up when writing something like 'toHex' to create a hexadecimal representation of a 'bytes':
  - Could append a 'bytes' created with 'bytes.format':

```
for byte in myBytes { asHex += b"%02xu".format(byte); }
```

  - Or, could use 'openMemFile', use 'writef' to output hex-formatted values, and then use 'readAll(bytes)':

```
var f = IO.openMemFile();
{
  var w = f.writer();
  for byte in myBytes { w.writef("%02xu", byte); }
}
var asHex = f.reader().readAll(bytes);
```

- Both have high overhead as compared to computing an ASCII value and appending that byte
  - Overhead comes from interactions with the I/O system and allocation overhead
    - Note: 'string.format' and 'bytes.format' are implemented through the I/O system
- Led to performance problems when using 'toHexString' from the Crypto package module

Appending to 'bytes' / 'string'
This Effort

- Added unstable methods to append any number of codepoints or numeric bytes to 'bytes'/'string':

```
proc ref string.appendCodepointValues(codepoints: int ...): void
proc ref bytes.appendByteValues(x: uint(8) ...): void
```

- Here is an example using these:

```
var myString: string, myBytes: bytes;
myString.appendCodepointValues(0x48, 0x69);                // appends "Hi"
myBytes.appendByteValues(0x54, 0x68, 0x65, 0x72, 0x65);    // appends "There"
writeln(myString, " ", myBytes);                           // outputs "Hi There"
```

- Also, added a method to convert a 'bytes' to hexadecimal since this is a common case

```
proc bytes.toHexadecimal(uppercase: bool = false, type resultType = bytes): resultType
```

Appending to 'bytes' / 'string'

Impact and Next Steps

Impact:

- **4,000x** speedup in a 'toHex' benchmark
- **13x** speedup in a user's application once 'Crypto.toHexString' was updated to use these

Next Steps: Make these methods stable & reduce the overhead of 'string.format'

Other Language Improvements

Other Language Improvements

For a more complete list of language changes and improvements in the 1.33 and 2.0 releases, refer to the following sections in the CHANGES.md file:

- New Language Features
- Language Feature Improvements
- Semantic Changes / Changes to the Chapel Language
- Deprecated / Unstable / Removed Language Features
- Language Specification Improvements
- Bug Fixes


Thank you
https://chapel-lang.org
@ChapelLanguage