Language Improvements

Chapel versions 1.21 / 1.22

April 9 / 16, 2020

chapel_info@cray.com
chapel-lang.org
@ChapelLanguage

CRAY
a Hewlett Packard Enterprise company

CRAY CHAPEL

Chapel 2.0

Chapel 2.0: An upcoming release in which we…

…commit to not breaking core language features

…switch to semantic versioning

- Our primary goal for this release cycle was to produce a candidate for Chapel 2.0
  - approached the effort with a list of features to address, major and minor
  - successfully addressed the vast majority of them

- Due to this focus, this release's notes are even more language-centric than usual

Chapel 1.21 vs. 1.22

- One key Chapel 2.0 focus area related to changing from 1- to 0-based indexing
- Not surprisingly, this can involve significant changes for some Chapel codes
- For this reason, we did two back-to-back releases a week apart:

  Chapel 1.21: our typical semi-annual release, with improvements of all kinds

  Chapel 1.22: Chapel 1.21, but with changes related to 0-based indexing

Outline

Online RST / HTML Language Specification

Online Spec: Background

- Since 2006, the Chapel language specification has been a LaTeX / PDF doc
  - PDF originally seemed like the ideal format for an authoritative document

- Over time it had become less attractive:
  - Editing LaTeX feels fairly heavyweight today
  - PDF was not well-integrated with newer online docs:
    - Couldn't search both at the same time
    - Couldn't link into the spec from outside it
      - e.g., difficult to refer a user to a specific section
    - Challenging to maintain links from the spec to the online docs

Statements 92

At a given place where control flow exits a block, the in-scope `defer` statements and the local variables will be handled in reverse declaration order. Handling a `defer` statement consists of executing the contained clean-up action. Handling a local variable consists of running its deinitializer if it is of record type.

When an iterator contains a `defer` statement at the top level, the associated clean-up action will be executed when the loop running the iterator exits. `defer` actions inside a loop body are executed when that iteration completes.

The following program demonstrates a simple use of `defer` to create an action to be executed when returning from a function:

Example (defer1.chpl).

```
class Integer {
  var x:int;
}
proc deferInFunction() {
  var c = new unmanaged Integer(1);
  writeln("created ", c);
  defer {
    writeln("defer action: deleting ", c);
    delete c;
  }
  // ... (function body, possibly including return statements)
  // The defer action is executed no matter how this function returns.
}
deferInFunction();
```

produces the output

```
created {x = 1}
defer action: deleting {x = 1}
```

The following example uses a nested block to demonstrate that `defer` is handled when exiting the block in which it is contained:

Example (defer2.chpl).

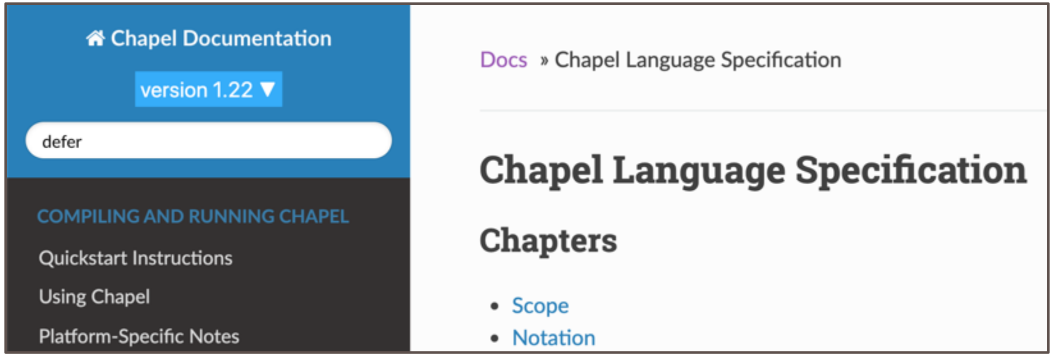
```
class Integer {
  var x:int;
}
proc deferInNestedBlock() {
  var i = 1;
  writeln("before inner block");
  {
    var c = new unmanaged Integer(1);
    writeln("created ", c);
    defer {
      writeln("defer action: deleting ", c);
      delete c;
    }
    writeln("in inner block");
    // note, defer action is executed no matter how this block is exited
  }
  writeln("after inner block");
}
deferInNestedBlock();
```

produces the output

Online Spec: This Effort, Impact

This Effort:

- Converted spec to RST, the same format as our other docs
- Integrated into online documentation page



Impact:

- Searching the online docs now finds entries in the spec
- Spec formatting matches the rest of the documentation

Online Spec: Next Steps

- Review online spec for formatting issues / improvements
- Review spec content for Chapel 2.0 readiness

Module / Namespace Improvements

Modules / Namespaces: Background

- Chapel has historically been a bit slack about namespaces
  - Design and implementation were focused on modest-scale codes
  - For larger-scale / more disciplined programming, approach was weak

Modules / Namespaces: This Effort

- Improving modules / namespaces was a major theme for Chapel 1.21
  - Private 'use' by default
  - Available-by-default symbols
  - Renaming 'use'd modules
  - 'import' statements
  - Relative 'use' and 'import' statements
  - Submodules in different files
  - Implicit module warnings
  - Parent module visibility

Private ‘use’ by default

Private 'use': Background

- Traditionally, 'use' statements have been 'public' (transitive) by default

```
module M1 {
  use BigInteger;    // traditionally interpreted as 'public use'
  …
}
module M2 {
  use M1;            // using 'M1' gives 'M2' access to 'BigInteger.*' as well
  var b: bigint;     // therefore, this was OK
}
```

- As a result, modules tended to unintentionally "leak" symbol names

Private ‘use’: This Effort

- Changed ‘use’ to be ‘private’ by default in user modules and standard modules

```
module M1 {
  use BigInteger;     // now interpreted as 'private use'
  …
}
module M2 {
  use M1;             // using 'M1' no longer gives 'M2' access to 'BigInteger.*'
  var b: bigint;      // error: 'bigint' undeclared
}
```

Private ‘use’: Impact

- Namespaces are more contained by default
- Codes that were relying on ‘use’ being public by default must be updated
  - by switching to ‘public use’ if the symbols had intentionally been exposed:

```
module M1 {
  public use BigInteger;
}
```

  - by adding additional ‘use’ (or ‘import’) statements to client code otherwise:

```
module M2 {
  use M1, BigInteger;
  var b: bigint;
}
```

Private ‘use’: Next Steps

- Update ‘use’ within internal modules to be ‘private’ as well
  - Doing so was non-trivial and didn’t make it into this release
  - However, current behavior should not affect user code (see next topic)
  - Nonetheless, making this change will…

    …improve the organization of internal modules

    …simplify the compiler slightly

Available-by-Default Symbols

Default Symbols: Background

- Chapel has traditionally made many symbols available by default
  - Some by design

```
writeln("hello, world!");
var infile = open("data.dat").reader(iokind.native);
```

  - Some due to historical lack of 'private use' feature

```
const myint = infile.read(c_int);    // 'c_int' shouldn't be available, but is
```

Default Symbols: This Effort (IO module)

- Reduced the number of symbols that are available by default
  - Default IO symbols reduced to just ‘write’, ‘writeln’, and ‘writef’

```
writeln("hello, world!");    // OK, writeln() available by default
```

  - Others now require ‘use IO;’

```
use IO;        // ‘use IO’ now required for more involved IO like this:
var infile = open("data.dat").reader(iokind.native);
```

  - **Rationale:** keep simple cases simple, more involved cases explicit

Default Symbols: This Effort (unintended modules)

- Reduced the number of symbols that are available by default
  - In internal modules, explicitly added ‘private’ to ‘use’s of standard modules

```
module String {
  private use SysCTypes;      // 'private' added in this release
  …
}
module DefaultSparse {
  private use RangeChunk;     // 'private' added in this release
  …
}
```

  - Previously, these caused the modules’ symbols to be available unintentionally

Default Symbols: Impact

- Fewer symbols are made available to user programs by default
- Explicit ‘use’ required for code that had relied on such default-available symbols
  - e.g., ‘use IO’ for programs doing non-trivial IO
  - e.g., ‘use SysCTypes;’ for programs that had previously relied on the auto-use

```
use IO;               // explicit ‘use’ now required for nontrivial IO
use SysCTypes;        // explicit ‘use’ now required to access SysCTypes.*
var infile = open("data.dat").reader(iokind.native);
const myint = infile.read(c_int);
```

Default Symbols: Next Steps

- Continue reviewing the set of symbols made available by default
- Continue improving ‘use’ statements within internal modules
  - untangle web of ‘use’s between internal modules

Renaming Used Modules

Rename-in-use: Background, This Effort

Background:

- Prior to this effort, could only rename submodules via ‘use’ of its parent

```
use OuterMod only InnerMod as Foo;  // Renames ‘InnerMod’ to ‘Foo’
```

- Couldn’t rename top-level modules at all

This Effort:

- Added support for renaming a module when it’s used

```
use OuterMod as Bar;  // Now supported!
```

Rename-in-use: Impact, Next Steps

Impact:

- Can now rename every module
- No longer stuck typing long module names for qualified access

Next Steps:

- Implement ability to ‘use’ module and disable qualified access (issue #15457)

```
use Mod as _;
writeln(Mod.x);  // Wouldn’t work, not enabled by this ‘use’
writeln(x);      // Would still work, enabled by this ‘use’
```

Import Statements

Import Statements: Background

- ‘use’ statements enable access to a module’s symbols from another module

```
use MyModule;
```

- But ‘use’ statements have been imprecise
  - Default behavior brought every visible symbol into scope
    - However, could limit the symbols brought in with ‘except’ and ‘only’ lists
  - Design focused on “programming in the small” scenarios

- Users desired a feature for more precise access of module symbols
  - One better suited for maintaining large-scale software
  - Ideally, without breaking current code

Import Statements: This Effort

- We designed and implemented the ‘import’ statement as an alternative
  - Simplest form enables qualified access to the symbols in a module:

```
import MyModule;
writeln(MyModule.sym1);  // Enabled by the 'import'
writeln(sym1);           // Not enabled, won't work
```

  - This was previously only achievable with “empty” use statements, e.g.

```
use MyModule only;
use MyModule except *;
```

(as part of this effort, we replaced such ‘use’s in libraries with ‘import’)

Import Statements: Accessing Module Contents

- Can also enable unqualified access to a single symbol within a module:

```
import MyModule.sym1;
writeln(sym1);              // Enabled by the 'import'
writeln(MyModule.sym1);     // Not enabled by the 'import'
```

- Or multiple symbols within a module:

```
import MyModule.{sym1, sym2, sym3};
```

- Neither of these options was available previously
  - ‘use’ statements always enabled qualified access in addition to unqualified

Import Statements: Renaming

- Modules that are imported can be renamed:

```
import MyModule as Foo;
writeln(Foo.sym1);         // Enabled by the 'import'
writeln(sym1);             // Not enabled by the 'import'
writeln(MyModule.sym1);    // Not enabled by the 'import'
```

- As can symbols that are imported for unqualified access:

```
import MyModule.sym1 as x;  // or:
import MyModule.{sym1 as x, sym2 as y};
```

Import Statements: Nested Modules

- Nested modules must be named using their parent modules…

```
module OuterMod {
  import InnerMod;                 // error: looks for top-level module 'InnerMod'
  import OuterMod.InnerMod;        // OK: names module starting from top-level
  writeln(InnerMod1.sym1);
  module InnerMod { var sym1 = …;  }
}
```

- Or after being made available by another 'import' or 'use'

```
  use OuterMod;         // makes 'OuterMod's symbols available
  import InnerMod;      // 'InnerMod' visible due to 'use OuterMod'
```

- Or using a keyword-based relative path (see next section)

Import Statements: Public / Private

- ‘import’ statements can be declared ‘public’ or ‘private’
  - Default is ‘private’
    - as with ‘use’, reduces unintentional leaking of names
  - ‘public’ means symbols brought in are *re-exported*

```
module Mod {
  public import OtherMod;
}
module ThirdMod {
  import Mod.OtherMod;  // ‘OtherMod’ acts like a submodule of ‘Mod’
}
```

Import Statements: Impact

- The ‘import’ statement supports module access in a more precise manner
  - Its default behavior minimally extends the scope
- It also enables new functionality:
  - Can re-export symbols
  - Can bring symbols in for unqualified access without enabling qualified access
- The ‘use’ statement is still available

Import Statements: Next Steps

- Extend 'import' to support multiple expressions in a single statement

```
import Mod1.{a, b}, Mod2.{x, y}; // Should this be allowed?
```

  - See issue #14971 and #15583

- Allow 'import' statements to refer to private symbols in parent modules
  - See issue #15308

- Enable re-exporting for 'use' statements
  - See issue #15282

Relative ‘use’ and ‘import’ statements

Relative Use: Background, This Effort

Background:

- ‘use’ statements could specify any module in scope
- When multiple modules share a name, could lead to confusion or errors

This Effort:

- Allowed ‘this’ and ‘super’ to specify path from current module

```
use this.Submodule;            // Uses module defined within current module
use super.SiblingModule;       // Uses module defined in parent module
```

- Supported for ‘import’ statements as well:

```
import this.Submodule;
import super.SiblingModule;
```

Relative Use: Impact, Next Steps

Impact:

- Origin of relatively used modules is much more obvious to the reader
- This style of 'use' makes code more robust to later changes
    - If dependency defines another module with same name, won't conflict
- Simplest way to 'import' local submodules
    - Otherwise, must specify a path from a top-level module

Next Steps:

- Fix bug where 'import' can't reference parent module's symbols via 'super'

```
import super.nonModSym; // Possible with full path to parent instead
```

  - See issue #15309

Submodules in Different Files

Submodules: Background

- Nested modules always had to be defined within the text of a parent module
  - This resulted in some large source files
  - It also discouraged deep or wide module hierarchies

- Python has a strategy for specifying submodules through a file hierarchy
  - It's a bit burdensome in ways, but seemed like a good design starting point

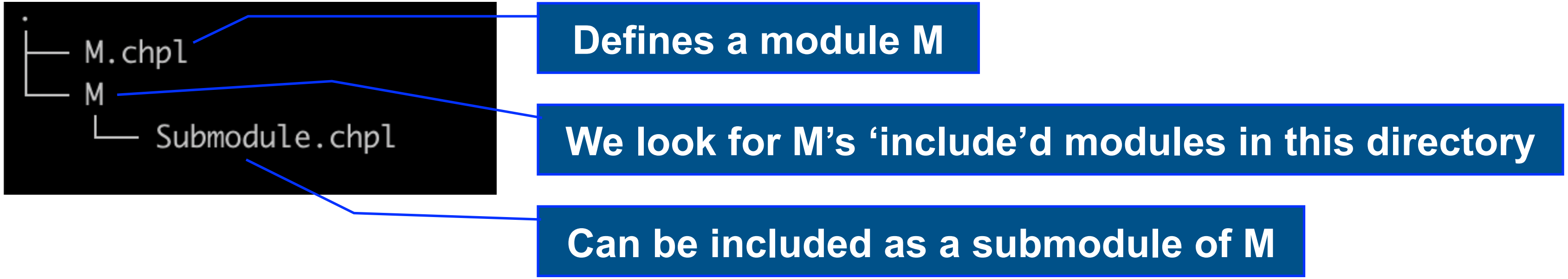
Submodules: This Effort

- Added 'include' keyword to declare a module in another file as a submodule

```
include module Submodule;
```

  - Submodules declared this way are treated like normal submodules

- File with submodule must be in a subdirectory with the parent module's name
  - the module names must match their base filenames



Submodules: Public vs. Private

- Included submodules can be declared ‘public’ or ‘private’
  - When ‘public’, makes Submodule available to modules that ‘use’ this module
  - When ‘private’, Submodule is unavailable to modules that ‘use’ this module

```
include private module Submodule;
```

  - Default is ‘public’, as with typical module declarations

Submodules: Public vs. Private

- Modules declared ‘public’ in their file can be included either publicly or privately

M.chpl

```
include private module Submodule;
```

M/Submodule.chpl

```
[public] module Submodule { … }
```

Submodule is private to M

- Modules declared ‘private’ in their file cannot be included publicly

M.chpl

```
include public module Submodule;
```

M/Submodule.chpl

```
private module Submodule { … }
```

Submodule was declared private, can’t be changed here

Submodules: Next Steps

- Finalize design
  - Relax current naming conventions?
  - Permit submodules to live in other places?

- Lean on implementation more
  - We believe it is correct, but need more experience with it

Implicit Module Warnings

Implicit Module Warnings

Background:

- File-scope statements (other than ‘module’ decls) generate an implicit module
  - Any module declarations in the same file become submodules
  - E.g. for a file ‘Foo.chpl’:

```
var b: int;                  // This line causes the creation of a module 'Foo'
module TopLevel { … }        // This is a submodule of 'Foo' now!
```

- It can be difficult to notice when and why this happens

This Effort:

- Improved warning for cases that can lead to this confusion
  - Mixtures of file-scope module declarations and other code in one file

Parent Module Visibility

Parent Module Visibility: Background

- Traditionally, child modules have been able to refer to their parents' contents:

```
module Parent {
  var p = 33;
  module Child {
    writeln(p);            // rationale: 'p' is lexically visible, so I can name it
    writeln(Parent.p);     // same for 'Parent'
  }
}
```

Parent Module Visibility: This Effort

- Reconsidered this rule due to the new ability to store submodules in distinct files:
  - Parent.chpl:

```
module Parent {
  include module Child;
}
```

  - Parent/Child.chpl:

```
module Child {
  writeln(p);            // problem: What the heck is 'p'?
  writeln(Parent.p);     // Or 'Parent'?
}
```

Parent Module Visibility: This Effort

- Decided that sub-modules must ‘use’ or ‘import’ their ancestors to refer to them
  - Parent.chpl:

```
module Parent {
  include module Child;
}
```

  - Parent/Child.chpl:

```
module Child {
  import Parent; import Parent.p;
  writeln(p);
  writeln(Parent.p);
}
```

Parent Module Visibility: This Effort

- Decided that sub-modules must ‘use’ or ‘import’ their ancestors to refer to them
  - (even when in a single file, for consistency)

```
module Parent {
  module Child {
    import Parent;
    import Parent.p;

    writeln(p);
    writeln(Parent.p);
  }
}
```

Parent Module Visibility: Status, Impact, Next Steps

Status:

- Updated modules and tests to reflect these rules

Impact:

- Generally resulted in cleaner code
- Analogous to requiring top-level modules to import/use one another
  - introduced in Chapel 1.20
- Can reduce compile-time when sub-modules don’t have need of ancestors

Next Steps:

- Get more experience with this change

Initialization and Deinitialization

Initialization & Deinitialization

- Variable initialization and deinitialization are Chapel language concepts
- Records can supply 'init', 'init=', and 'deinit' methods to be called by compiler

- This section discusses improvements in this area:
  - Split initialization is now supported and improves 'out' intent
  - Copy elision covers more cases and improves 'in' intent
  - Deinitialization points for temporary variables are improved

- The goal is to address known issues and allow language stabilization in this area

Split Initialization

Split Initialization: Background

- Split initialization is a feature requested by users
  - in response to challenges combining error handling and non-nilable classes
- The idea is to declare a variable in one statement and initialize it in another, e.g.:

```
var x;
x = 1;
```

- Historically such code would be a compilation error

```
error: Variable 'x' is not initialized and has no type
```

- With split initialization, the second statement 'x = 1' is initializing 'x'

Split Initialization: This Effort

- Added support for split initialization to the language
- Split initialization applies to variables declared with no initialization expression
- Compiler searches forward from variable declarations for applicable assignments
  - could be assignment or a variable passed to an 'out' intent formal argument
- If no applicable assignment is found
  - default initialize the variable if possible
  - otherwise, issue a compiler error

Split Initialization: Approach

- The compiler looks forward from a declaration to find applicable assignments

```
var x;
// ... compiler searches the statements that follow for applicable assignments
```

- Searches forward for applicable assignments within:
  - block declarations { }
  - 'try' and 'try!' blocks
  - conditionals
- An applicable assignment is the first mention of the variable that sets it
  - can be an assignment statement like 'x = 1'
  - can be a function call passing to an 'out' intent formal

Split Initialization: Example 1

```
record R { ... }
var a: R;
{
  a = new R();    // split initialization
}

var b: R;
writeln(b);
b = createR();    // not split init (not first mention), so 'b' is default inited above
```

Split Initialization: Conditionals

- Split initializations within conditionals allow for multiple applicable assignments
  - both branches of the conditional must initialize the variable, or
  - one branch initializes the variable and the other returns or throws
- If a conditional initializes multiple variables, order must match on both branches

Split Initialization: Example 2

```
var c: R;
if option {
  c = new R();    // split initialization
} else {
  c = new R(1);    // split initialization
}

var d: R;
if option {
  d = new R();    // not split initialization, d is default-initialized above
}
```

Split Initialization: Example 3

```
var e: R;
if option {
  e = new R();    // split initialization
} else {
  return;
}

var f: R;
try {
  f = createR();    // split initialization
}
```

Split Initialization: Status

- The now compiler supports split initialization for all major local symbol types:
  - 'var', 'const', 'type', 'ref', 'const ref'

- Split initialization for module-level variables is also supported
  - except for 'config' variables

Split Initialization: Impact on 'out' intent

- Used split initialization to improve the handling of the 'out' intent

```
var x: R;
setArg(x);                    // split init of 'x'
proc setArg(out argument: R) {
  argument = g();             // split init of 'argument'
}
```

- As a result, the above example contains 0 copies or assignments
  - in earlier releases it would perform 2 assignments

Copy Elision

Copy Elision: Background

- *Copy elision* is a language feature to avoid copy initialization in certain cases
- A form of *copy elision* already existed in 1.20 for nested call expressions:

```
record R { ... }
proc makeRecord() {
  return new R();
}
var a = globalRecord;    // copy initialization
var x = makeRecord();    // move initialization, not copy initialization
```

Copy Elision: This Effort

- Extended *copy elision* to cover more cases
- In particular, elide the copy when the source of the copy initialization is:
  - a local non-reference variable that is not mentioned again

- Once a copy is elided, the source variable becomes dead
- Compile-time analysis provides compiler errors for many erroneous cases

Copy Elision: Approach

- Like split initialization, the analysis searches forward from variable declarations
- Searches for copy initializations that are the last mention of the source variable
  - within block declarations { }, conditionals, try blocks, and try! blocks
  - but not within loops, on-statements

Copy Elision: Example 1

```
proc elideCopy() {
  var x = makeRecord();
  var y = x;     // copy elided because 'x' is not used again
}

proc noElideCopy() {
  var x = makeRecord();
  var y = x;     // copy is not elided because 'x' is used again
  writeln(x);    // 'x' used here
}
```

Copy Elision: Example 2

```
proc elideCopyBothConditional() {
  var x = makeRecord();
  var y;    // split initialization below
  if option {
    y = x;
  } else {
    y = x;
  }
  // copy is elided because 'x' is not used after the copy
  // (in either branch of the conditional or after it)
}
```

Copy Elision: Example 3

```
proc copyElisionError() {
  var x = makeRecord();
  ref refX = x;
  var y = x;          // copy elided because 'x' is not used again
  writeln(refX);      // use of dead variable - error reported by compiler
}
```

Copy Elision: Impact on 'in' intent

- Historically, passing an 'in' intent argument to another function caused a copy:

```
proc f(in a) { ... }
prog g(in b) {
  f(b);    // always copied here
}
```

- Now, copy elision applies to such cases to remove the copy
  - because the argument 'b' is not mentioned again after passing it to 'f'

Deinitialization Points

Deinit Points: Background

- Historically, used a simple rule to decide when to deinitialize a local variable
  - deinitialized at end of enclosing block
  - in reverse declaration order
- Led to confusing behavior for the following example:

```
{
  var f = IO.opentmp();
  var A = [1,2], B = [0,0];
  f.writer().write(A);    // data buffered in temporary channel
                          // until it is flushed when deinited at end of block
  f.reader().read(B);     // reads 0s because buffer not yet written
}
```

Deinit Points: This Effort

- Revisited rule for when to deinitialize a local variable to avoid this issue
  - also to be more similar to other languages
- Local user variables always deinitialized at the end of their containing blocks
- Temporaries for nested call expressions have new behavior
  - Deinitialized at end of block when contained in an initialization expression

```
var x = f(g());    // temporary storing g() deinitialized at end of block
```

  - Deinitialized at end of statement otherwise

```
f(g());    // temporary storing g() deinitialized at end of this statement
```

Deinit Points: This Effort

- With split init, initialization order can differ from declaration order
  - Adjusted deinitialization order of locals to be reverse initialization order

```
{
  var a;
  var b;
  // declaration order: a, b

  b = makeRecord();
  a = makeRecord();
  // initialization order: b, a
} // deinitialization order: a, b
```

Deinit Points: Impact

- I/O example behavior is less surprising

```
{
  var f = IO.opentmp();
  var A = [1,2], B = [0,0];
  f.writer().write(A);   // data buffered in temporary channel
                         // which is deinited and flushed at end of statement

  f.reader().read(B);    // reads 1, 2 as expected
}
```

Init / Deinit Summary

Initialization & Deinitialization: Status, Next Steps

Status:

- Split init, copy elision, and deinit point are implemented and documented
  - These features address known issues and user requests
- Language rules in this area are expected to be stable from this point

Next Steps:

- Fix array initialization and copy initialization to no longer always default-init
- Fix deinitialization order for split-initialized module-scope variables

String and Bytes Improvements

String and Bytes: Background

- Chapel 1.20 took some steps towards having UTF-8 strings
  - Adjustments to the 'string' type to support Unicode data
    - Codepoint-based indexing, iteration, and length measurement
    - Can still opt-in to byte-based methods

  - 'bytes' type was added to store arbitrary bytes and support string operations
    - Does not have to store Unicode data
    - Potentially better performance on ASCII text

String and Bytes: This Effort

- Require Chapel strings to store valid Unicode data

- 'bytes' improvements
  - Add missing features for 'bytes'
  - Adjust standard/package modules to support 'bytes' where applicable

- Stabilize 'string' and 'bytes' interfaces

String Validation

String Validation: Unicode Validation

- Strings are required to store UTF-8 encoded data
  - String literals are validated at compile-time

```
var s = "Invalid byte: \xff";   // compiler error: "Invalid string literal"
```

0xFF cannot appear in a UTF-8 sequence

  - Dynamically created strings (e.g., through I/O) are validated at runtime

```
var s: string;
readerChannel.readstring(s);   // throws SystemError if not a valid string
```

  - Strings created with factory functions are also validated at runtime
    - Unless you are creating from another string, which is assumed to be valid

- These rules do not apply to 'bytes'

String Validation: Non-UTF-8 Filenames

- POSIX standard does not limit filenames to UTF-8
  - Many operating systems do (e.g. SLES, MacOS, Cray)
  - Some do not (e.g. Ubuntu, Windows)

- How can filenames be handled where we enforce UTF-8 strings?
  - Option 1: Use 'bytes'
    - Less convenient; no implicit conversion between 'string' and 'bytes'
  - Option 2: Escape non-UTF-8 sequences and store them in 'string'
    - Not a breaking change, interface stays the same

String Validation: Escaped Strings

- Similar to Python's "surrogate escapes"
  - Prepend '0xDC' to individual bytes of illegal sequences
  - Encode this 2-byte codepoint in UTF-8 and store in the string

- 'createStringWithNewBuffer' now provides this functionality:

```
var s = createStringWithNewBuffer(c"\xff",   // not UTF-8
                                  policy=decodePolicy.escape);
```

- 'FileSystem', 'Path' and 'IO' modules are adjusted to use this strategy
  - Provides portability using Chapel strings across different systems

String Validation: 'string.encode()'

- 1.20 provided 'bytes.decode()'
  - Only way to create a 'string' from a 'bytes'
  - Optional argument sets the behavior when data is not UTF-8
    - 1.21 adds a new option to decode with escapes

- 1.21 adds 'string.encode()'
  - Without argument, it is a synonym for casting to 'bytes' from a 'string'
  - Optional argument sets the behavior when 'string' contains escapes
    - Either reconstruct original data, or copy as-is

String Validation: 'encode'/'decode' samples

- Reconstruct escaped byte sequences

```
s.encode(policy=encodePolicy.unescape)  // b"\xff"
```

- Or, keep them as-is

```
s.encode(policy=encodePolicy.pass)  // UTF-8 for 0xDCFF as 'bytes'
```

- A 'bytes' can also be decoded with escaping

```
b.decode(policy=decodePolicy.escape);
```

Bytes Improvements

Bytes Improvements: General Features

- 1.21 completes the 'bytes' features that were missing in 1.20
  - 'param' bytes can be defined and used the same way as 'param' strings
  - 'bytes' values can be compared using '<', '>', '<=', '>='
  - 1-byte 'bytes' can be converted to 'uint(8)' using 'bytes.toByte()'
  - 'bytes' can be used as associative domain indices
  - 'bytes.format()' can be used to create a new bytes

    ```
    b"Name: %s ID: %i\n".format(name, id)    // returns a new 'bytes'
    ```

  - 'bytes.this()' accepts 'byteIndex' to support generic programming with strings
  - 'bytes' can be cast to enum

Bytes Improvements: Regular Expressions

- Added support for 'bytes'-based regular expressions

```
var re = compile(b"[[:alnum:]]+@[[:alpha:]]+\\.edu");
var line: bytes;
while readChan.readline(line) {
  const m = line.match(re);
  if m then
    writeln(line[m]);   // prints emails ending in "edu"
}
```

would fail if 'string' was used and the file wasn't UTF-8

Bytes Improvements: Regular Expressions

- 'regexp' is now generic and can be based on 'string' or 'bytes’
  - As of 1.21, 'regexp' defaults to 'regexp(string)'
    - Provides backward-compatibility
    - Generates a deprecation warning

- In the next releases 'regexp' will not default to 'regexp(string)'
  - Type must be used explicitly if a fully instantiated type is meant

Bytes Improvements: Other Libraries

- I/O expanded to support 'bytes'
  - 'channel.readline()' now accepts 'bytes' arguments
  - 'channel.readbytes()' was added similar to 'channel.readstring()'

- ZMQ module now supports 'bytes' messages

```
import ZMQ;
var myMsg = b"Some \xff arbitrary \xff data";
ZMQ.send(myMsg);
var theirReply = ZMQ.recv(bytes);
```

  - e.g., Arkouda messages between client and server now use 'bytes'

Interface Stabilization for Strings & Bytes

String / Bytes Interfaces: Indexing Updates

- Indexing and iterating over 'bytes'
  - 'bytes.this()' and 'bytes.these()' now return/yield 'uint(8)' instead of 'bytes'

```
writeln(b"Chapel"[0]);    // prints "67" not "C"
```

- New methods and iterators for generic programming with 'string' and 'bytes'
  - 'item()' and 'items()' on 'string' and 'bytes' return the same types

```
writeln(b"Chapel".item(0));    // prints "C"
writeln("Chapel".item(0));     // prints "C"
```

  - For 'string' they are just synonyms for 'this' and 'these'

String / Bytes Interfaces: Indexing Summary

CRAY
a Hewlett Packard Enterprise company

was 'bytes' in 1.20

	Return or Yield Type	
proc / iter name	**string**	**bytes**
this / these	string	uint(8)
byte / bytes	uint(8)	uint(8)
item / items	string	bytes
codepoint / codepoints	int(32)	N/A

added in 1.21

this == item, these == items for string

String / Bytes Interfaces: Stabilization

- Argument name adjustments
  - Some arguments were named 's' in both 'string' and 'bytes' interfaces
    - They are renamed to 'x' to be more type-neutral
    - e.g. factory functions, 'join()'
  - "policy" arguments are renamed to 'policy' instead of 'errors'
    - 'errors' was inherited from Python's similar interface, but it is misleading
    - e.g. 'bytes.decode()' and newly-added 'string.encode()'

- 'string' vs 'bytes' comparisons are deprecated
- 'string' initializers that were deprecated in 1.20 are removed

String / Bytes Summary

String and Bytes: Impact and Next Steps

Impact:

- 'string' can only store valid UTF8-encoded data
- 'bytes' is more versatile and can be used instead of strings in many places
- More consistent and type-neutral interface in 'string' and 'bytes'

Next Steps:

- Performance analysis and improvements
- Reduce uses of 'c_string'
- Ensure that environment is set for using UTF-8 during program startup

Class Improvements

Class Improvements: Background

- Classes have gone through major changes in recent releases:
  - managed classes introduced in Chapel 1.18, improved in 1.19–1.20
  - nilable classes introduced in Chapel 1.20
- Have continued to have some rough edges that needed sanding down

Class Improvements: This Effort

- Managed Class Improvements
- Better Checking for Non-Nilable Types
- Collection x Type Compatibility Study
- The ‘!’ Operator
- Prototype Modules and Nilability
- Accessing Type and Param Fields on Nilable Class Types

Managed class improvements

Managed Classes: Background, This Effort

Background: Previously, compiler parsed the following cases the same:

```
new owned X();  // typical usage
new owned(X);   // creating an owned to manage an unmanaged instance
new owned X;    // should have been an error
```

This Effort:

- Adjusted the parser to differentiate cases and made 'new owned X' an error:

```
new owned X;        // now an error
```

- Deprecated 'new owned(X)' and added a clearer alternative:

```
new owned(X);       // now deprecated
owned.create(X);    // clearer replacement
```

Managed Classes: Status, Impact

Status: 'owned.create' and 'shared.create' are implemented and documented

- both accept unmanaged class instances
- both accept nilable or expiring owned
- 'shared.create' accepts shared class instances

Impact:

- 'new owned X' is now an error as expected
- Clearer way to express creating an owned/shared with an unmanaged

Better Checking for Non-Nilable Types

Non-Nilable: Background, This Effort

Background: Non-nilable class types had incomplete checking in 1.20

- There were type checking problems for:
  - default initialization of arrays of non-nilable
  - resizing arrays of non-nilable
  - associative arrays of non-nilable
  - ownership transfer from non-nilable

This Effort: Address these problems with compile-time and run-time checking

Non-Nilable: Rectangular Arrays

- Now check for missing rectangular array initialization

```
var A: [1..1] borrowed C; // oops! non-nilable element not initialized
                          // now compilation error
```

- Now check for invalid resize of non-nilable rectangular array

```
var D = {1..1};
var B: [D] owned C = [new C(),];
D = {1..2}; // oops! new non-nilable element B[2] not initialized
            // now a runtime error
```

Non-Nilable: Associative Arrays

- Now check for associative arrays of non-nilable

```
var D: domain(int) = {1, 2};
var A: [D] borrowed C = myClass.borrow();    // now compilation error
```

  - Error is currently necessary because:
    - Adding a new key to D would cause A to try and default-initialize a value
    - But, non-nilable classes have no default value
  - The error applies to *any* associative array of non-nilable
    - Hope to relax this check in the future
  - Sparse arrays should have a similar check (and do on master, but not 1.22)

Non-Nilable: This Effort

- Added compile-time checks for use of dead non-nilable owned

```
var x = new owned C();
var y = x;    // ownership transfer copy initialization which leaves 'x' dead
writeln(x); // oops! use of dead value 'x'
error: mention of non-nilable variable after ownership is
transferred out of it
```

- If 'x' is not mentioned again, the ownership transfer copy is elided and it works:

```
var x = new owned C();
var y = x;    // elided copy because x is not mentioned again
writeln(y);
```

- See also section on copy elision

Non-Nilable: Impact, Status, Next Steps

Impact: Addressed several missing checks in the type system

- Programs cannot continue to rely on incorrect behavior

Status: Many known type system problems with non-nilable are addressed

Next Steps:

- Explore extending support for arrays of non-nilable to include:
  - associative arrays
  - sparse arrays
  - resizable rectangular arrays

Collection x Type Compatibility Study

Collections x Types: Background

Background:

- Many interacting features have been introduced in recent releases
  - Managed classes introduced in Chapel 1.18
  - Nilability introduced in Chapel 1.20
  - Lists, maps, and sets introduced in Chapel 1.20
- The combination of all collection-type interactions had not been fully tested
- Users reported several compatibility issues between types and collections

Collections x Types: This Effort

This Effort:

- Wrote tests for matrix of combinations between collections and types
  - Goal is to understand what works, what does not, and why not
  - Note that this effort took place just prior to the release
- Improved error messages for cases that are not expected to work
- Investigated fixes for failing cases

Collections x Types: Testing Procedure

- Wrote minimal API tests for each collection and tested with many types
  - This is intended to catch obvious issues, not all edge cases
- To reduce the matrix size, tuples of shared class represents all tuple cases
  - Future work could investigate tuples of all types

Collections x Types: Status on 1.22

	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	✅	✅	✅	✅	✅	✅	✅	✅

<u>Key</u>

Working

Not yet working

🔷 Not expected to work

Collections x Types: Status (non-nilable tuples)

	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	✓	✓	✓	✓	✓	✓	✓	✓

These cases fail due to a default initialization bug for tuples

Key

Working

Not yet working

◆ Not expected to work

Collections x Types: Status (Lists)

	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

- Borrowed cases require some lifetime checking fixes
  - Now fixed on master (1.23 pre-release)
  - Error messages are clean in 1.21

Collections x Types: Status (Maps)

	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

- Borrowed cases require lifetime constraints to work
  - Error messages are clean in 1.21

Collections x Types: Status (Map of owned t)

	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

- Non-nilable owned types not currently expected to work
  - Challenges related to ownership and nilable
  - Error messages are clean in 1.21
  - May be supported in future releases

Collections x Types: Status (Sets of owned)

	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

- Owned types not expected to work
  - No way to test membership b/c set takes ownership
  - Error messages are clean in 1.21
  - Could potentially support these with restricted API

Collections x Types: Status (Sets)

	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

- Remaining failures due to issues w/ default assoc.
  - Most stem from default hashes not supporting classes
  - Most issues are now fixed on master

Collections x Types: Status (fixed-size arrays)

	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

- Non-nilable classes supported for init'd fixed-size arrays
  - No default value required if not resized

Collections x Types: Status (other arrays)

CRAY
a Hewlett Packard Enterprise company

	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

- Other array types not expected to support non-nilable
  - Resizing domains requires default values
  - Most cases give reasonable errors in Chapel 1.21

Collections x Types: Status on 1.22

	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

Collections x Types: Status on master

	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

Collections x Types : Next Steps

Impact:

- Collection and class type interactions are better understood
- Most combinations that are not expected to work have good error messages
- Several failing cases have been fixed on master since 1.22

Next Steps:

- Fix remaining failing cases
- Improve documentation on what types are supported
- Extend testing to more cases
  - Associative domains (similar to ‘set’)
  - Tuples of class types other than ‘(shared *, shared *)’

The '!' Operator

'!' Operator: Background

- '!' is a postfix operator that converts class values to the non-nilable variant
  - Halts when applied to 'nil' values if runtime checks are enabled
    - Checks are disabled by --no-nil-checks / --no-checks / --fast

```
var x: owned C?; // x.type == owned C?
var y = x!;      // y.type == borrowed C, halts if checks are enabled
```

  - Open question whether these checks should always be enabled
    - Ideally want checks always on, but has performance implications

'!' Operator: This Effort

- Explored performance implications of '!' always checking for 'nil'
  - Overhead was significant, 2x slowdown for PRK-Stencil and NAS-FT
- Added limited compiler optimization for '!' checks within conditionals
  - Ineffective for impacted applications, significant effort required to improve
- Manually optimized distributions
  - Used internal feature to force-unwrap nilables

'!' Operator: Status

- Identified two patterns where '!' is used in performance sensitive code
  - Code that needs a conditional guard where subsequent uses are non-nil
  - Code where implementation context guarantees non-nil
- With current language definition, always-on checks are too expensive
  - Believe this can be resolved with optional chaining and force-unwrapping

'!' Operator: Optional Chaining Background

- Conditional guard is required, but subsequent uses are non-nil

```
proc BlockArr.dsiAccess(idx) ref {
  if myLocArr != nil && myLocArr!.contains(idx) then
    return myLocArr!.dsiAccess(idx);
  return nonLocalAccess(idx);
}
```

- Repeated '!' uses after nil check impacts readably
  - And would hurt performance if '!' always had a nil check

'!' Operator: Optional Chaining Next Steps

- Want some sort of optional chaining like Swift

```
if myLocArr != nil && myLocArr!.contains(idx) then
  return myLocArr!.dsiAccess(idx);
```

=>

```
if let arr = myLocArr? && arr.contains(idx) then
  return arr.dsiAccess(idx);
```

'!' Operator: Force-Unwrap Nilable

- Code where implementation context guarantees non-nil

```
proc BlockArr.dsiLocalAccess(idx) ref {
  return myLocArr!.dsiAccess(idx);
}
```

- A runtime check introduces unacceptable overhead
  - Want a mechanism to force-unwrap a nilable with no runtime overhead
    - Believe this is sufficiently rare that it does not warrant syntax

```
return forceToNonNilable(myLocArr).dsiAccess(idx);
```

'!' Operator: Next Steps

- Mitigate '!' check overheads
  - Add support for optional chaining
  - Add ability to force unwrap a nilable with no runtime check
- Enable always-on '!' runtime checks

Prototype Modules and Nilability

Prototype Modules and Nilability

Background: In 1.20, prototype modules changed nilable class behavior

```
var x: owned C? = ...;
x.method(); // allowed in a prototype module but not production module
            // compiler automatically replaced 'x' with 'x!'
```

- This difference led to confusion

This Effort: Removed this difference for prototype modules

- Now the above code needs to be written

```
x!.method();
```

Impact: Easier to move code between prototype and non-prototype modules

- Now the only difference is the requirement to handle errors

Accessing Type and Param Fields on Nilable Class Types

Type Methods on Nilable: Background

- Previously required '!' to use a type/param field on a nilable type:

```
class C {
  param p;
}
type t = owned C(1)?;
t.p;    // produced internal error
t!.p;  // worked
```

Type Methods on Nilable: This Effort, Impact

This Effort: Adjusted compiler to no longer require '!' in this case:

```
class C {
  param p;
}
type t = owned C(1)?;
t.p;    // now works
```

- Also, deprecated '!' on types in favor of more explicit casts
  - since behavior of '!' is not obvious when applied to types

```
t!.p;  // compilation warning: ! on types is now deprecated
```

Impact: Easier to write generic code using nilable types

0- vs. 1-based Indexing

0-based Indexing: History Lesson, Part I

- Whether a language uses 0- or 1-based indexing is a question w/ no good answer
  - each approach has its benefits and proponents
- From its original design, Chapel strived to be index-neutral
  - e.g., ranges and rectangular domains require low and high bounds

```
const r = 1..10;     var A: [0..#n] real;
```

- However, we were unsuccessful at making Chapel completely index-neutral:
  - e.g., tuples, anonymous arrays, and string indexing from the outset:

```
var A = [1.2, 3.4];   // what does A[1] refer to?
var t = (1.2, 3.4);   // what does t(1) refer to?
…"brad"[1]…           // which letter does this refer to?
```

  - since then, bytes and lists have been introduced and have similar issues

0-based Indexing: History Lesson, Part II

- At the time of its design, Chapel was primarily focused on users of…

  …**C/C++:** 0-based …**Fortran:** 1-based

  …**Java:** 0-based …**Matlab:** 1-based

- This made the decision seem like a coin-toss, so we went with 1-based indexing
  - **Rationale:** most people count from 1, and we were striving for productivity

0-based Indexing: Background

- However, Chapel users also complained of seeming inconsistencies:
  - most notably, certain built-in arrays chose to count from 0:
    - Locales (rationale: HPC programmers count nodes from 0)
    - args to main() (rationale: argv[0] typically refers to executable name)
  - since these are arrays, they are arguably free to choose their low bound
    - yet, being built-in, they have been a source of confusion for users
- Meanwhile, most notable recent languages have used 0-based indexing:
  - Python, Rust, Swift, Go, …
- And, most early Chapel adopters have come from C/C++ or Python backgrounds
  - notably, despite being 1-based, Chapel has not attracted many Fortran users

0-based Indexing: This Effort

- We polled Chapel users about switching to 0-based indexing
  - Most said they would prefer it, if we were designing the language from scratch
  - Most were not terribly concerned about updating their existing Chapel code
  - Most expressed concern about the expected impact to other users
- We then decided to gauge the impact on our own code base:
  - internal, standard, package modules (~150 files, ~150,000 lines)
  - Chapel tests: (~12,000 source files, ~125,000 lines)
  - mason: (19 source files, ~6,000 lines)
- Also gauged the impact on:
  - CrayAI (19 files, ~3800 lines)
  - Arkouda (~39 files, ~12,000 lines)

0-based Indexing: This Effort

- Based on all this input, we decided to make the switch
  - given the push for Chapel 2.0, seemed like a “now or never” decision
  - though it would be impactful and annoying, decided it was worthwhile
- Given the impact, we decided to constrain it to its own release
  - Thus, Chapel 1.21 is our normal semi-annual release
  - Chapel 1.22 is essentially 1.21 with 0-based indexing
- This permits users to incrementally upgrade to 0-based indexing

0-based Indexing: What changed?

- Primary cases that switched to 0-based indexing:
  - tuples

```
var t = (1.2, 3.4);    // t(1) was 1.2; it's now 3.4, and t(0) is 1.2
```

  - strings, bytes

```
…"chapel"[1]…   …b"chapel"[1]…    // was "c"; it's now "h", and …[0] is "c"
```

  - arrays whose size is not defined by a range, domain, or array

```
var A = [1.2, 3.4];    // A[1] was 1.2; it's now 3.4, and A[0] is 1.2
var B = myIter();      // B was defined over domain {1..}; it's now over {0..}
```

  - lists

```
var l: list(int) = …;    //  l(1) was the first element in the list; it's now the
                         // second, and l(0) is the first
```

0-based Indexing: What changed?

- Secondary cases that switched to 0-based indexing:
  - varargs:

```
proc foo(arg...) { writeln(arg(1)); }    // 'arg' now counts from 0
foo(1.2, 3.4);      // used to print 1.2; it now prints 3.4 since varargs are tuples
```

  - tuple-oriented methods:

```
const D = {1..3, 1..5};
writeln(D.dim(1)); // used to print 1..3; it now prints 1..5 and .dim(0) prints 1..3
```

  - search-oriented methods:

```
…"chapel".find("z")…    // used to return 0; it now returns -1
```

  - field numbering:

```
myRecord.getField(1)    // used to return the first field; it now returns the second
```

0-based Indexing: What changed?

- Secondary cases that switched to 0-based indexing:
  - random streams:

```
myRandomStream.getNth(i)  // used to count from 1; it now counts from 0
```

- Other cases to be wary of:
  - untyped captures of ‘split()’ calls

```
var substrs = myString.split();  // capture an inferred-size array of strings
…substrs(1)…     // this used to refer to the 1st substring; it now refers to the 2nd
```

0-based Indexing: What's didn't change?

- Arrays whose size is not defined by a range, domain, or array

```
var C = [i in -1..1] foo(i);   // C's domain is still inferred to be {-1..1}…
var D = 2*C;                   // …as is D's
```

- Source file line numbers are still 1-based
  - **rationale:** because text editors are

```
vard answer = 42;    // test.chpl:1: syntax error: near 'answer'
```

0-based Indexing: Impact

- Code changes required in the Chapel repository were approximately as follows:

	Files modified	Lines of code modified
Tuple-related changes	~860 (7%)	~7300 (2.5%)
String/Bytes-related changes	~125 (1%)	~650 (0.2%)
Arrays	~125 (1%)	~570 (0.2%)
Lists	~40 (0.3%)	~156 (0.05%)

- Though lots of code needed to be updated, most changes were straightforward
  - bounds-checking at compile-time and execution-time caught most cases
  - automated testing helped find and fix others
  - only a minimal number of cases were truly tricky or laborious to track down

0-based Indexing: Next Steps

- Help users update their code and adjust to 0-based indexing
  - tips available online: https://chapel-lang.org/docs/1.22/language/evolution.html
- Update additional cases that ought to be:
  - 'Sort' module's keyPart() interface should probably use 0-based indexing
  - Check for any other interfaces that should be updated

Index-Neutral Features

Index-Neutral: Background

- Chapel has always had features supporting index-neutral programming:
  - '.domain' queries:

```
proc foo(A: [] ) {
  forall i in A.domain do …
}
```

  - de-tupling:

```
var (x,y,z) = myTuple;
foo(myTuple);
proc foo((x, y, z)) { … }
```

Index-Neutral: This Effort

- Updating files from 1- to 0-based indexing motivated new index-neutral features:
  - '.indices' queries on arrays, tuples, strings, bytes, lists,

was: `for i in 1..myCollection.size do …`

now: `for i in 0..#myCollection.size do …`

better: `for i in myCollection.indices do …`

  - loops over heterogeneous tuples

was: `for param i in 1..myCollection.size do …myTup(i)…`

now: `for param i in 0..myCollection.size-1 do …myTup(i)…`

better: `for t in myTup do …t…`

Index-Neutral: This Effort

- Updating files from 1- to 0-based indexing motivated new index-neutral features:
  - open-interval ranges:

was: `for i in cursor..myCollection.size do …`

now: `for i in cursor..myCollection.size-1 do …`

better: `for i in cursor..<myCollection.size do …`

  - '.first' / '.last' queries on enums:

```
enum color {red, green, blue};
```

was/now: `for c in color.red..color.blue do …`

better: `for c in color.first..color.last do …`

Index-Neutrality: Next Steps

- Continue looking for ways to support index-neutral programming in Chapel
  - inferred-size arrays:

```
var A: [1..] = myIter();    // assert 1-based indices, but not size nor eltType
```

  - array destructuring:

```
var [a, b, c, …] = MyArray;
var (a, b, c, …) = MyArray;
```

Other Changes for Chapel 2.0

Chapel 2.0: Other Changes to Scalar Types

- Made ‘<<’ well-defined for signed integers
- Made bad enum casts throw
- Changed the locale type to have value semantics
- Updated the atomic compareExchange API to match C/C++

Chapel 2.0: Other Changes to Aggregate Types

- Documented tuple semantics in language specification
- Required records to support ‘init=’ and ‘=’ or neither
- Added support for creating non-copyable records
- Added a default ‘<’ operator for records
- Fixed ability to use methods/fields of private types via instances
- Made ‘C’ a subtype of ‘C?’
- Made assignment overloads for classes illegal
- Started enforcing ‘override’ keywords for compile-time (type/param) methods
- Made Error classes store strings and preserve line numbers

Chapel 2.0: Other Changes to Interfaces

- Made ‘readThis’/‘writeThis’ throw
- Deprecated synonyms for ‘.size’ (‘.length’, ‘.numIndices’, ‘.numElements’)

Chapel 2.0: Deprecated Features

- C++-style names for deinitializers

```
proc ~C() …     ⇒    proc deinit() …
```

- ‘enumerated’ as a type class in favor of ‘enum’

```
proc foo(e: enumerated)    ⇒    proc foo(e: enum)
```

- support for spaces within type queries

```
proc foo(x: ? t) …    ⇒    proc foo(x: ?t)
```

Chapel 2.0: New –warn-unstable warnings

- Expected to evolve further as they receive more attention:
  - first-class functions
  - unions
- Features known to be buggy / ill-defined:
  - arrays with negative strides
- Future uncertain:
  - identifiers beginning with ‘chpl_’ or ‘_’
  - ‘new borrowed C()’
  - enums with duplicate integer values and semi-concrete enums
  - let statements

Chapel 2.0: Outstanding Issues

- Constrained generics (see Ongoing Efforts deck)
- Point-of-instantiation definition
- Impact of GPU support

Chapel 2.0: Features expected to evolve

- Parallel iterators
- User-defined domain map interface
- User-defined reduction / scan interface
- Array initialization (from default initialization + assignment to copy initialization)
  - affects arrays-of-records, though beneficially
- Skyline arrays
- Zippered iterations involving sparse / associative domains and arrays

```
forall (i,j) in (mySparseArr, myDenseArr) do …
```

- Capturing iterators in type-inferred variables

```
var A = myIter();
```

For More Information

For a more complete list of library-related changes in the 1.21 and 1.22 releases, refer to the following sections of the CHANGES.md file:

- Syntactic/Naming Changes
- Semantic Changes / Changes to the Chapel Language
- New Features
- Feature Improvements
- Deprecated / Unstable / Removed Language Features
- Standard Library Modules
- Error Messages / Semantic Checks

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?

chapel_info@cray.com
@ChapelLanguage
chapel-lang.org

a Hewlett Packard Enterprise company

cray.com
@cray_inc
linkedin.com/company/cray-inc-/