Hewlett Packard Enterprise

CHAPEL 1.31/1.32 RELEASE NOTES: LIBRARY IMPROVEMENTS

Chapel Team

June 22, 2023 / September 28, 2023

OUTLINE

IMPROVEMENTS TO STANDARD DISTRIBUTIONS

DISTRIBUTION IMPROVEMENTS

- Distributions as Records
- Distribution Factory Methods
- Redistributing Block Arrays
- Optimized Swaps

CONVERTING DISTRIBUTIONS TO RECORDS

DISTRIBUTIONS AS RECORDS

Background

- Historically, distributions have been implemented as ‘class’ types in Chapel (e.g., ‘Block’ is a class)
  - This made them something of an outlier in Chapel’s standard libraries
    - Most library-based types are records, for simplicity: no need to worry about ownership types, nilability, etc.

- When declaring named distributions, best practice has been to wrap them with a ‘dmap’ record type
  - Gave them value semantics, providing symmetry with domains and arrays

```
var myDist = new dmap(new Block(boundingBox={1..4, 1..8}));
```

- The ‘dmap’ type has always been a bit unpopular and obscure
  - In most cases, it could be avoided by just distributing domains directly

```
var Dom = {1..n, 1..n} dmapped Block({1..n, 1..n});
```

  - Yet, being able to declare and reuse named distributions remains valuable
    - amortizes overheads, guarantees alignment

```
var Dom1 = {1..n, 1..n} dmapped myDist,
    Dom2 = {0..n+1, 0..N+1} dmapped myDist;
```

DISTRIBUTIONS AS RECORDS
This Effort

- Decided to work toward deprecating the ‘dmap’ type to avoid being stuck with it in Chapel 2.0

- Changed standard distribution types from classes into records
  - Provides the convenience and consistency of a value type
  - Removes the need for the ‘dmap’ wrapper type

- Renamed distribution types—e.g., ‘Block’ is now named ‘blockDist’
  - Renaming has several benefits:
    - matches standard module style guide for record naming (camelCase)
    - clarifies the type’s role (e.g., ‘block’ is a very general term)
    - avoids using potentially common identifiers (e.g., ‘block’ is frequently used for various unrelated things)
    - improves symmetry with the module type (i.e., the ‘BlockDist’ module defines the ‘blockDist’ type)
  - Note that old names still work within standard code patterns, but generate a deprecation warning

DISTRIBUTIONS AS RECORDS
Status and Impact

Status:

- Applied changes in previous slide to all standard multi-locale distribution modules:
  - BlockDist, CyclicDist, StencilDist, ReplicatedDist, PrivateDist, HashedDist, BlockCyclicDist, DimensionalDist2D
- Single-locale layouts have yet to be updated
  - DefaultDist, CS

Impact:

- The ‘dmap’ type is no longer required to declare new distribution values
- Code involving distributions is now a bit more straightforward:
  - e.g.,

```
var myDist = new dmap(new Block(boundingBox={1..4, 1..8}));
```

would now be written:

```
var myDist = new blockDist(boundingBox={1..4, 1..8});
```

DISTRIBUTIONS AS RECORDS
Next Steps

Short-term:

- Convert standard layouts to records as well
- Deprecate the ‘dmap’ type

Medium-term:

- Look for additional opportunities for refactoring to enable code re-use and minimize boilerplate
- Improve documentation for creating distributions with a “how to” guide

Longer-term:

- Convert the standard domain map API from a convention to a set of standard interfaces

DISTRIBUTION FACTORY METHODS

DISTRIBUTION FACTORY METHODS
Background

- There has been a longstanding desire to replace the 'dmapped' keyword with new syntax
  - Like the ‘dmap’ type, the ‘dmapped’ keyword and syntax have not been very popular or memorable

- As design progresses, existing distribution factory methods provide a stable alternative

DISTRIBUTION FACTORY METHODS
This Effort

- Marked 'dmapped' syntax as unstable
  - Factory methods are a stable alternative for 'blockDist', 'cyclicDist', and 'stencilDist', e.g.,

```
/* unstable: */ var dom = {1..n} dmapped blockDist({1..n});
/* stable: */   var dom = blockDist.createDomain(1..n);
```

- Improved and unified factory methods on 'blockDist', 'cyclicDist', and 'stencilDist'
  - Added an instance-method overload of 'createDomain', e.g.,

```
var dom = myBlockDist.createDomain(1..n);
```

    – Rationale: without 'dmapped', this is currently the only stable way to have multiple domains share a single distribution

  - Added an optional 'targetLocales' argument to 'createDomain' and 'createArray' factory methods, e.g.,

```
var dom = blockDist.createDomain({1..n}, targetLocales=myLocales);
var arr = blockDist.createArray({1..n}, int, targetLocales=myLocales);
```

  - Added unstable overloads of 'createArray' that accept various expressions to initialize the array, e.g.,

```
var A1 = blockDist.createArray({1..5}, int, 1);                 // [1, 1, 1, 1, 1]
var A2 = blockDist.createArray({1..5}, int, [1, 2, 3, 4, 5]);   // [1, 2, 3, 4, 5]
var A3 = blockDist.createArray({1..5}, int, [i in {1..5}] i*i); // [1, 4, 9, 16, 25]
```

DISTRIBUTION FACTORY METHODS

Impact and Next Steps

Impact:

- Unified factory method interfaces across our most stable distributions
- Expanded functionality for a stable alternative to 'dmapped'

Next Steps:

- Extend factory methods to other distributions as part of stabilizing them
- Design alternate syntax to directly replace 'dmapped' [#23128, #23328, #23331]

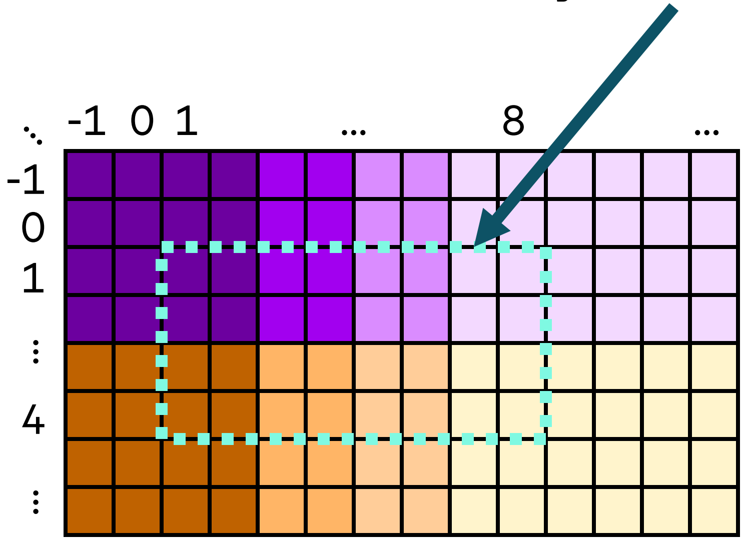
REDISTRIBUTING BLOCK ARRAYS

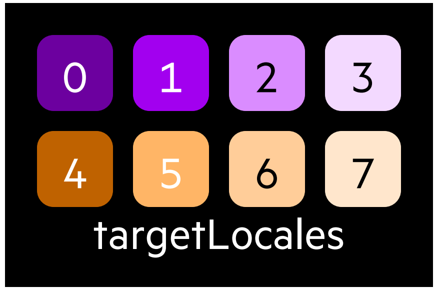
REDISTRIBUTING BLOCK ARRAYS

Background

- Block-distributed arrays are characterized by a “bounding box”
  - specifies which d-dimensional indices are block-distributed across locales (as evenly as possible)
  - indices outside the bounding box map to the same locale as their closest interior neighbor

```
var myDist = new blockDist(boundingBox={1..4, 1..8});
```





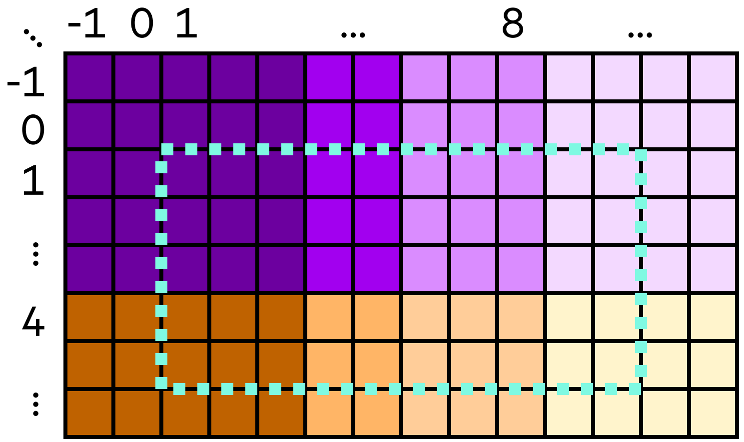
- Traditionally, this box could not be changed once a distribution object was created

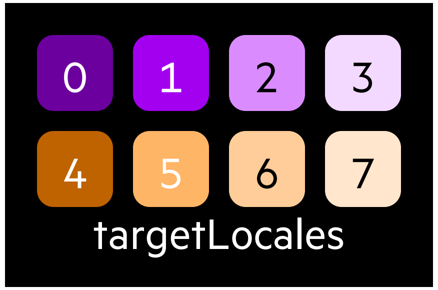
REDISTRIBUTING BLOCK ARRAYS

This Effort

- Added initial support for redistributing a block distribution, as long as no arrays need to be preserved
  - supports the common case of wanting to change the distribution shortly after declaration, before arrays exist

```
var myDist = new blockDist(boundingBox={1..4, 1..8});
myDist.redistribute({1..5, 1..10});    // or:  myDist = new blockDist({1..5, 1..10});
```





- Notably, no compiler or language changes were required to add this capability

REDISTRIBUTING BLOCK ARRAYS
Status and Next Steps

Status:

- Users can now change a Block distribution's mapping before any domains or arrays are declared over it
- They can also redistribute Block-distributed arrays, as long as no data needs to be preserved
  - the current best practice is to:
    - deallocate any arrays over the distribution by making their domains empty (e.g., 'Dom = {1..0};')
    - redistribute the block distribution
    - re-allocate the arrays according to the new distribution by re-assigning the domains to their desired sizes
  - changing a distribution in other ways may result in undefined behavior for its domains and arrays
- We now have a proof-by-example that Chapel can support redistribution, as anticipated

Next Steps:

- Add the ability to preserve array values when redistributing a block distribution
- Consider adding the ability for a non-initialized Block distribution to use its first domain as its bounding box

```
var Dom = {1..n, 1..n} dmapped new blockDist();    // note the lack of a 'boundingBox' argument
```

- Consider extending support for redistribution to other distributions
- Consider renaming 'boundingBox' argument before 2.0?

OPTIMIZED SWAP FOR CYCLIC-/STENCIL-DISTRIBUTED ARRAYS

OPTIMIZED ARRAY SWAP
Background, This Effort, and Status

Background: Chapel 1.23 added an array swap optimization for default- and Block-distributed arrays

```
var A, B: [1..n] real;
A <=> B;    // optimized this to use a pointer swap rather than a deep copy (and similarly for Block-distributed arrays)
```

This Effort: Extended this optimization to Cyclic- and Stencil-distributed arrays

Status: Cases like the following are now optimized to use a pointer swap as well:

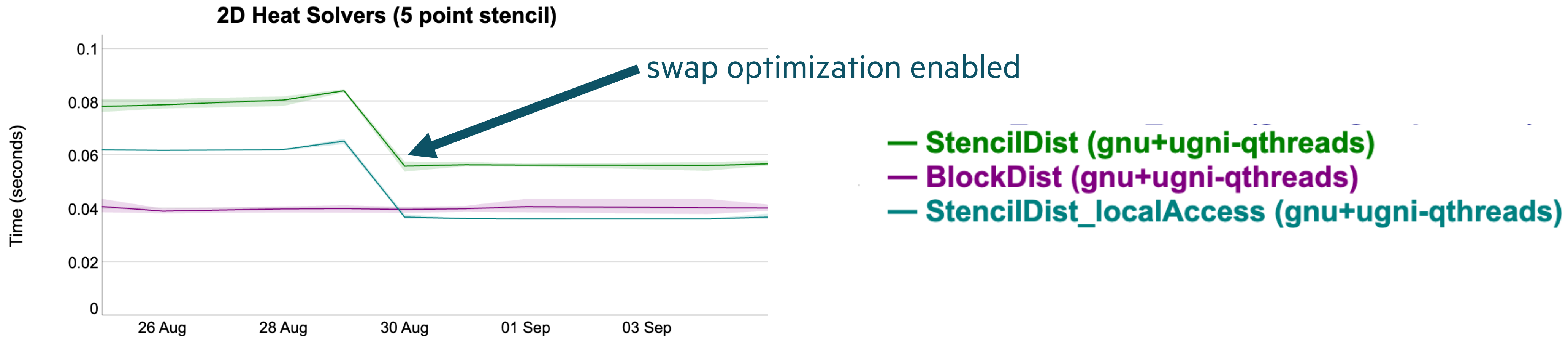
```
var CycDom = cyclicDist.createDomain({1..n, 1..n});
var C, D: [CycDom] real;
C <=> D;

var StencilDom = stencilDist.createDomain({1..n, 1..n});
var E, F: [StencilDom] real;
E <=> F;
```

OPTIMIZED ARRAY SWAP

Impact and Next Steps

Impact:

- Reduced time required to swap between Cyclic- or Stencil-distributed arrays
  - e.g., the following heat solver computations utilize array swaps between time steps:



Next Steps:

- Continue seeking out and addressing cases where Block-distributed arrays outperform Cyclic and Stencil
- As other distributions are stabilized, look for additional opportunities to apply this optimization
- Look into ways to refactor this optimization to simplify applying it to new distributions, and for code re-use

‘C_PTR’ IMPROVEMENTS

'C_PTR' IMPROVEMENTS

Background

- The Chapel 'c_ptr' type represents a C pointer within Chapel
  - Used primarily for C interoperability — 'c_ptr(T)' corresponds to C's 'T*'
  - Also used for pointers within Chapel, which are not otherwise exposed
  - Acquired either from calling extern C code, or via 'c_ptrTo':

```
extern proc myExternFunc(): c_ptr(c_int);        // extern declaration to call C function
var myPtr: c_ptr(c_int) = myExternFunc();        // a 'c_ptr' from C

var x: int = 5;
var myOtherPtr: c_ptr(int) = c_ptrTo(x);         // a 'c_ptr' entirely within Chapel
```

‘C_PTR’ IMPROVEMENTS
Background

- ‘c_ptrTo’ has had special behavior on arrays
  - Returns a pointer to the first element instead of the array’s metadata
- ‘c_ptr’ also had some limitations / non-orthogonalities:
  - Had to use a separate ‘c_void_ptr’ type to represent a ‘void*’, with casting/implicit conversion to ‘c_ptr’
  - Could be dereferenced to mutate the pointee
    - No way to represent a const pointer ‘const T*’
    - Couldn’t create a ‘c_ptr’ to a const variable via ‘c_ptrTo’
  - Could cast between ‘c_ptr’s of different pointee types without regard for C’s strict aliasing rules

‘C_PTR’ IMPROVEMENTS
This Effort

- Extended the value-based ‘c_ptrTo’ behavior on arrays to additional types
  - ‘string’ and ‘bytes’: Returns a ‘c_ptr(c_uchar)’ to the start of the underlying buffer
  - Class types: Returns a ‘c_ptr(void)’ to the heap-allocated instance of the class variable
  - Behavior transition controlled by compile-time ‘-scPtrToLogicalValue’ flag

- Added simpler ‘c_addrOf’ procedure that avoids the special behavior above for all types
  - Logically corresponds to C’s address-of operator ‘&’

```
use CTypes;
class Foo {}
var myFoo = new owned Foo();      // similar behavior with shared, unmanaged, etc.
writeln(c_addrOf(myFoo));         // stack address of pointer to heap-allocated object
writeln(c_ptrTo(myFoo));          // heap address of the Foo instance

// create "another" Foo, pointing to the same instance
var anotherFoo = (c_ptrTo(myFoo):unmanaged Foo?)!;
```

'C_PTR' IMPROVEMENTS
This Effort

- Replaced 'c_void_ptr' with 'c_ptr(void)'
  - Still prevents dereferencing
- Added 'c_ptrConst' type, like 'c_ptr' but with const pointee
  - Acquired via new 'c_ptrToConst', or external procedures
  - Special behavior above also applies to 'c_ptrToConst'

```
const oldStr: string = "foo";    // 'c_ptrTo(oldStr)' would yield "error: const actual is passed to 'ref' formal"
var newStr: string = "bar";
extern proc strcpy(dest: c_ptr(c_uchar), src: c_ptrConst(c_uchar));
strcpy(c_ptrTo(newStr), c_ptrToConst(oldStr));

var x : int = 5;
var mutablePtr = c_ptrTo(x);
mutablePtr.deref() += 1;            // ok
var constPtr = c_ptrToConst(x);
constPtr.deref() += 1;              // error: cannot assign to const variable
```

- Added warning for 'c_ptr' casts that violate C's strict aliasing rules

‘C_PTR’ IMPROVEMENTS
Impact and Next Steps

Impact:

- New ‘c_ptrTo’ functionality provides useful behavior in more cases
  - ‘c_ptr’s to ‘string’ and ‘bytes’ values more closely correspond to C behavior
  - Clarifies distinction between ’c_ptr’s to class heap instances, and to memory-management stack structures
- ‘c_ptr(void)’ unifies behavior and implementation with other ‘c_ptr(T)’s, less special-casing
- Can now represent C const pointers (‘const T*’)
  - Previously, had to (incorrectly) disregard constness in extern C function signatures with const pointers
  - Allows creating ‘c_ptr’s to ‘const’ Chapel variables
- Prevents unintentional undefined behavior via ‘c_ptr’ casts between pointee types

Next Steps:

- Explore techniques to mitigate pitfall of creating invalid ‘c_ptr’s across locales
- Consider separate types for C interoperability and user-facing memory buffer [#16797]

CHAPEL 2.0 LIBRARY STABILIZATION

CHAPEL 2.0 LIBRARY STABILIZATION
Background and Status

Background:

- Chapel 2.0 is an upcoming release in which core language and library features will be considered stable
  - *Stable:* Going forward, all changes will be backwards-compatible
  - Users should be able to depend on anything not noted as 'unstable' to continue working through all 2.X releases
    - Such features are noted as unstable in the documentation and/or will trigger warnings when using '--warn-unstable'
- Our primary focus has been on standard library stabilization

Status In Numbers:

- 39 modules reviewed
- 35 modules stabilized
- 10 standard modules that we've decided not to stabilize before Chapel 2.0:
  - CommDiagnostics, Memory[.Diagnostics], GMP, DynamicIters, VectorizingIterator, Help, GPU, GpuDiagnostics, Random, Heap

CHAPEL 2.0 LIBRARY STABILIZATION

1.30 Status

	Builtins	ChplConfig	List	Map	Set	FileSystem	IO	Path	Reflection	Types	BigInteger	Math/AutoMath	Random	Collectives	CTypes	Subprocess	Sys	SysBasic	SysError	Regex	Time	Version	String / Bytes	Ranges	Domains	Arrays	Shared / Owned	Errors	MemMove	Locales	Sync/Single	Atomics	BitOps
1.28	✓ Vetted	Review Started	Review Started	Progress	Review Started	Review Started	Progress	✓ Vetted	Progress	Progress	Progress	Progress	Review Started	Progress	Progress	✓ Vetted	✓ Vetted	Progress	✓ Vetted	Progress	Progress	Progress	Progress	Progress	Progress	Progress	Progress	Review Started	Review Started	Review Started	Review Started	Review Started	
1.29	✓ Vetted	Review Started	Review Started	Progress	Review Started	Progress	Progress	✓ Vetted	Progress	✓ Vetted	Progress	Progress	Review Started	Progress	Progress	✓ Vetted	✓ Vetted	✓ Vetted	✓ Vetted	Progress	Progress	✓ Vetted	Progress	Progress	Progress	Progress	Progress	Progress	Progress	✓ Vetted	Review Started	Review Started	
1.30	✓ Vetted	Review Started	Progress	Progress	Review Started	Progress	Progress	✓ Vetted	Progress	✓ Vetted	Progress	Progress	Review Started	Progress	Progress	✓ Vetted	✓ Vetted	✓ Vetted	✓ Vetted	✓ Vetted	Progress	✓ Vetted	Progress	Progress	Progress	✓ Vetted	Progress	Progress	✓ Vetted	✓ Vetted	Review Started	Review Started	

✓ **Vetted** | **Progress** | **Review Started** | X **Not for 2.0**

CHAPEL 2.0 LIBRARY STABILIZATION

1.32 Status

	Builtins	ChplConfig	List	Map	Set	FileSystem	IO	Path	Reflection	Types	BigInteger	Math/AutoMath	Random	Collectives	CTypes	Subprocess	Sys	SysBasic	SysError	Regex	Time	Version	String / Bytes	Ranges	Domains	Arrays	Shared / Owned	Errors	MemMove	Locales	Sync/Single	Atomics	BitOps
1.30	✓	Review Started	Progress	Progress	Review Started	Progress	Progress	✓	Progress	✓	Progress	Progress	Review Started	Progress	Progress	✓	✓	✓	✓	✓	Progress	✓	Progress	Progress	Progress	✓	Progress	Progress	✓	✓	Review Started	Review Started	
1.31	✓	Review Started	Progress	Progress	✓	Progress	Progress	✓	Progress	✓	Progress	Progress	X	✓	Progress	✓	✓	✓	✓	✓	Progress	✓	Progress	Progress	Progress	✓	Progress	Progress	✓	✓	Progress	Progress	✓
1.32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ **Vetted** | **Progress** | **Review Started** | X **Not for 2.0**

LIBRARY STABILIZATION OUTLINE

- IO
- Math/AutoMath
- BigInteger
- Collection Types
- Errors
- Collectives
- Time
- FileSystem
- Reflection
- CTypes
- ChplConfig
- BitOps

I/O MODULE

I/O SERIALIZERS

I/O SERIALIZERS OUTLINE

- Background
- High-Level Usage
- Custom Type Serialization
- Implementing (De)Serializers
- Status and Next Steps

BACKGROUND

I/O SERIALIZERS
Background

- Historically, non-default I/O formats consisted of a fixed set of options embedded in the ‘IO’ module
  - Adding new formats presented difficulty and was not user-facing

- The ‘iostyle’ record could be used to tweak various details of reading or writing
  - For example, setting the starting/ending character for a string
  - Supported over a dozen settings to support different formats

- Binary I/O was generally configured using ‘iostyle’ or ‘iokind’ types
  - ‘iokind’ indicated endianness, could only be set when the channel was created, and was poorly named

- JSON and “Chapel syntax” formats supported by ‘%jt’ and ‘%ht’ format specifiers
  - These were hard-coded into the ‘readf’/’writef’ implementations
  - Behind the scenes, used a mixture of ‘iostyle’ options and specialized implementations

I/O SERIALIZERS
Background

- For user-defined types, the ‘IO’ module invoked ‘readThis’ and ‘writeThis’ methods
  - For reading, required an initialized value to already exist
  - An example ‘writeThis’ method:

```
proc MyRecord.writeThis(f: fileWriter(?)) {
  f.writeln(this.id, ": ", this.data);
}
```

- These methods could be compiler-generated with somewhat flexible default behavior
  - For example, "print all fields in declaration order"
  - Provided *basic* support for the “default”, JSON, or “Chapel syntax” formats

- However, user-defined ‘readThis’/’writeThis’ methods were not so flexible
  - Supporting built-in formats (e.g., JSON) required using esoteric ‘iostyle’ settings
  - Even implementations for types in the standard library were difficult to write and maintain

I/O SERIALIZERS
This Effort

- Goals for an alternative way to choose how types are written:
  - Do not limit options to fixed set defined in standard ‘IO’ library
  - Make it possible for users to add other formats (e.g., YAML, Protobuf, etc)
  - Make it easy to write a custom I/O method for a user-defined type once, and have it work with multiple formats

- This new feature will apply to the ‘write’, ‘writeln’, ‘read’, and ‘readln’ methods on ‘fileReader’/’fileWriter’
  - Will also be invoked by ‘readf’/’writef’ using new ‘%?’ format specifier

- With this feature, finally deprecate ‘iostyle’ and ‘iokind’
  - As well as ‘%jt’ and ‘%ht’

I/O SERIALIZERS
This Effort

- In 1.32, we introduce a new way to choose how types are read/written: Serializers and Deserializers
  - Or, for brevity, "(De)Serializers"

- 'fileWriter' and 'fileReader' have an associated serializer or deserializer
  - Unless specified, the default (de)serializer will be used, which implements the existing default behavior

- A (De)Serializer must implement an API to be usable
  - To be enforced by Chapel interfaces as they mature

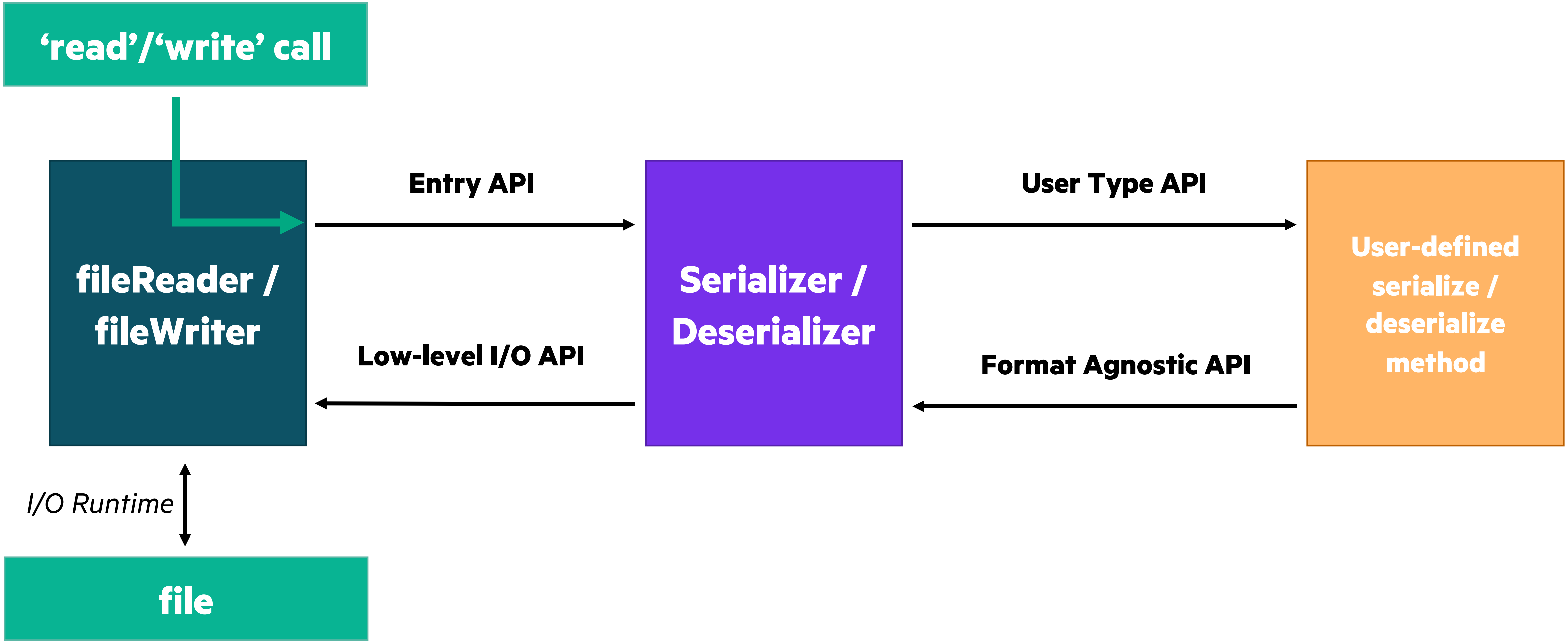
- Can be chosen when creating readers or writers, and can be changed on the fly afterward

- 1.32 provides 'default', 'binary', and 'JSON' (De)Serializers
  - With package modules for 'YAML' and "Chapel syntax" (De)Serializers

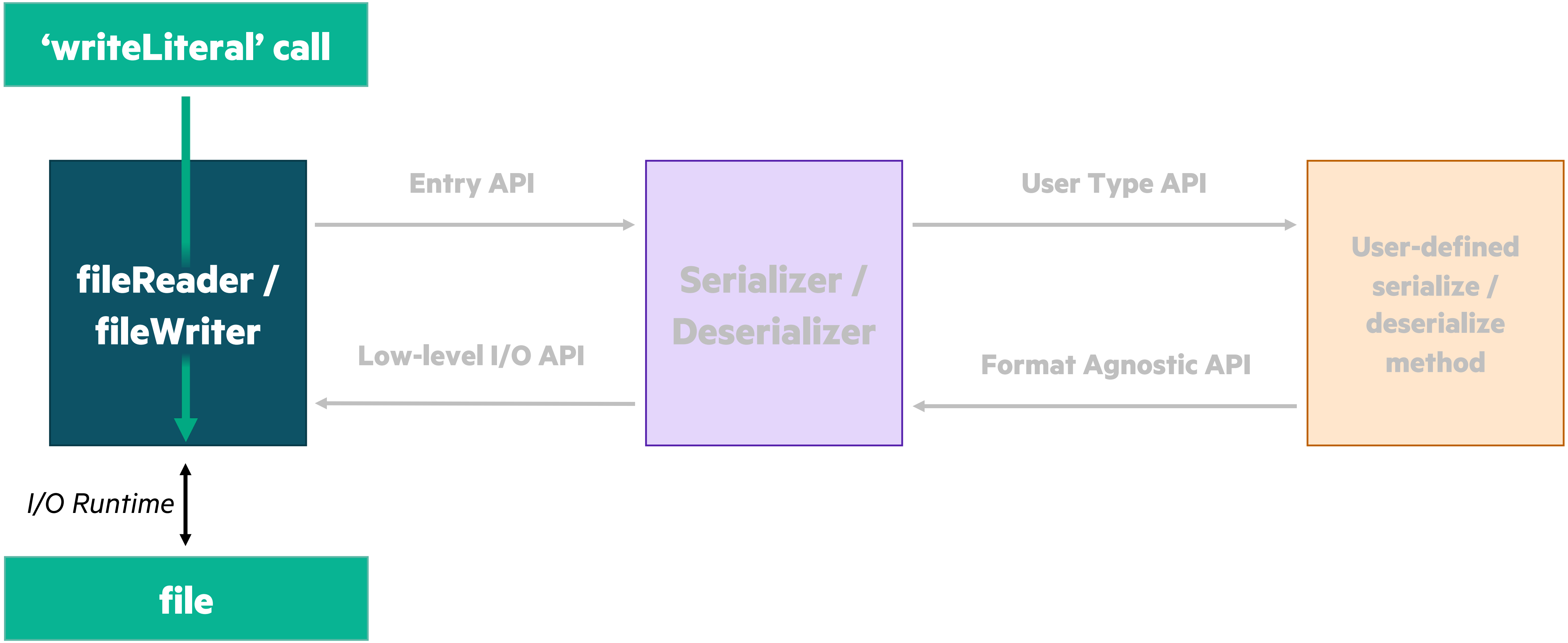
SERIALIZER API DESIGN
High-Level API Overview

- The ‘read’/’write’ methods hand off control to (De)Serializers
- (De)Serializers invoke user-defined ‘serialize’ and ‘deserialize’ methods when available
- ‘serialize’/’deserialize’ methods can use a format-agnostic API to comply with multiple formats
- Internally uses lower-level methods on ‘fileWriter’ and ‘fileReader’ to read/write specific characters
  - E.g., ‘writeLiteral’, ‘readByte’, etc.
  - These low-level methods do not go through the (De)Serializers API

SERIALIZER API DESIGN



SERIALIZER API DESIGN



HIGH-LEVEL USAGE

HIGH-LEVEL USAGE

Creating fileReader/fileWriter with (De)Serializers

- The ‘fileReader’ and ‘fileWriter’ types can be created with a specific Serializer or Deserializer
  - Otherwise, use ‘defaultSerializer’ or ‘defaultDeserializer’ from ‘IO’ module
  - Selected by optional ‘serializer’ or ‘deserializer’ arguments in ‘file.reader’, ‘file.writer’, ‘openReader’, or ‘openWriter’

- For example, consider a sample “data.json” file with a single JSON object:

```
{ "name": "Bob" }
```

- We can easily read this file into a suitable record in the following example:

```
use IO, JSON;
record R {
  var name: string;
}
var jsonReader = openReader("data.json", deserializer = new jsonDeserializer());
var r = jsonReader.read(R);
writeln(r);    // in ‘default’ format: (name = Bob)
```

HIGH-LEVEL USAGE

(De)Serializer Instances in fileReader/fileWriter

- ‘fileReader’/‘fileWriter’ have ‘.deserializer’/‘.serializer’ methods to access current instance
  - This ability exists in case a particular (De)Serializer provides additional non-standard methods for users

- The ‘serializerType’ and ‘deserializerType’ fields support queries and specialization:

```
// Allow any non-locking fileWriter
proc myFunction(writer: fileWriter(false, ?))
// Specific overload for JSON
proc myFunction(writer: fileWriter(false, serializerType=jsonSerializer))
```

- The ‘withSerializer’ and ‘withDeserializer’ methods allow for “changing” the format on the fly
  - These methods return an alias to the current ‘fileReader’/’fileWriter’ that will always point to the same file offset
  - These methods accept either a value or a type that can be default-initialized, for brevity

```
stdout.writeln("JSON output is:");    // ‘stdout’ uses the default format
stdout.withSerializer(jsonSerializer).writeln(myObj);
```

HIGH-LEVEL USAGE

Example: Mixed Format Binary File

- As an example, read a binary file with a little-endian 'int', a big-endian 'real', and a little-endian 'int'

- Users can configure their readers/writers when created:

```
use IO;    // brings in 'binaryDeserializer'
var little = new binaryDeserializer(ioendian.little);
var littleReader = myFile.reader(deserializer=little);
var myInt = littleReader.read(int);
```

- Can also adjust format from an existing reader/writer:
  - Here, 'bigReader' is an alias of 'littleReader' with the same offset in the file, but reads in big-endian

```
var big       = new binaryDeserializer(ioendian.big);
var bigReader = littleReader.withDeserializer(big);
var bigReal   = bigReader.read(real);
```

  - After that read, 'littleReader' shares the same offset in the file as 'bigReader'

```
var littleInt = littleReader.read(int);
```

CUSTOM TYPE SERIALIZATION

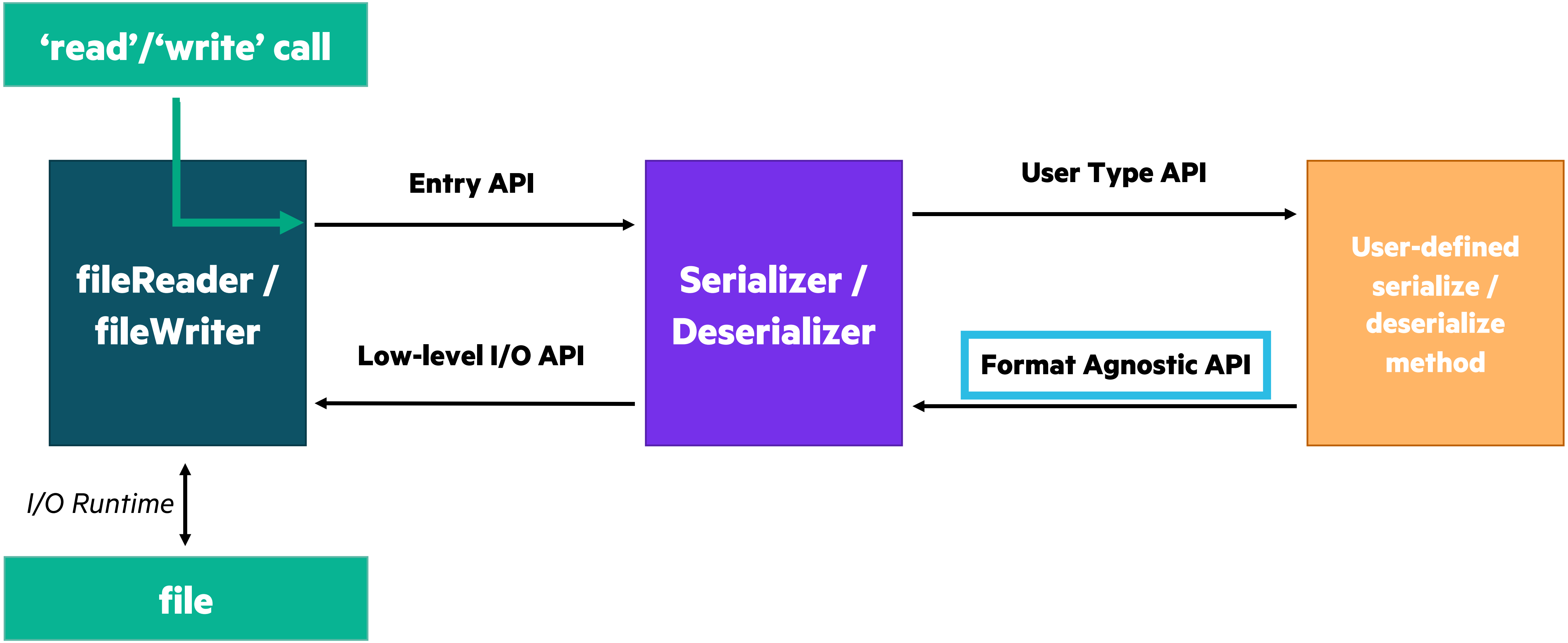
CUSTOM TYPE SERIALIZATION
The API

- The (De)Serializers API can be broken into roughly three pieces
  1. Methods called by the 'IO' module to hand off control to a (De)Serializer (relevant for (De)Serializer authors)
  2. Methods a (De)Serializer can invoke on user types to allow for customized I/O
  3. Methods a user-defined type can invoke on a (De)Serializer to perform format-agnostic I/O

- (De)Serializers support format-agnostic I/O for several kinds of abstract types
  - For example, many formats support their own notion of a "List" or "Map"
  - A portion of the API is devoted to each kind of abstract type

- See the IO Serializers technote for full details of the API

FORMAT-AGNOSTIC API



FORMAT-AGNOSTIC API
Methods on Serializers

- Serializers provide six ‘start’ methods to begin serializing a kind of type
  - Type-kinds: Class, Record, Tuple, Array, List, Map

- Each ‘start’ method takes a ‘fileWriter’ and returns an object with methods for the specific type-kind
  - Each ‘start’ method also accepts a ‘size’ argument, for example to represent a number of fields or elements

Class	Record	Tuple	Array	List	Map
startClass	startRecord	startTuple	startArray	startList	startMap
writeField	writeField	writeElement	writeElement	writeElement	writeKey
startClass*			startDim		writeValue
			endDim		
endClass	endRecord	endTuple	endArray	endList	endMap

* note: second ‘startClass’ exists to support inheritance

FORMAT-AGNOSTIC API

Methods on Deserializers

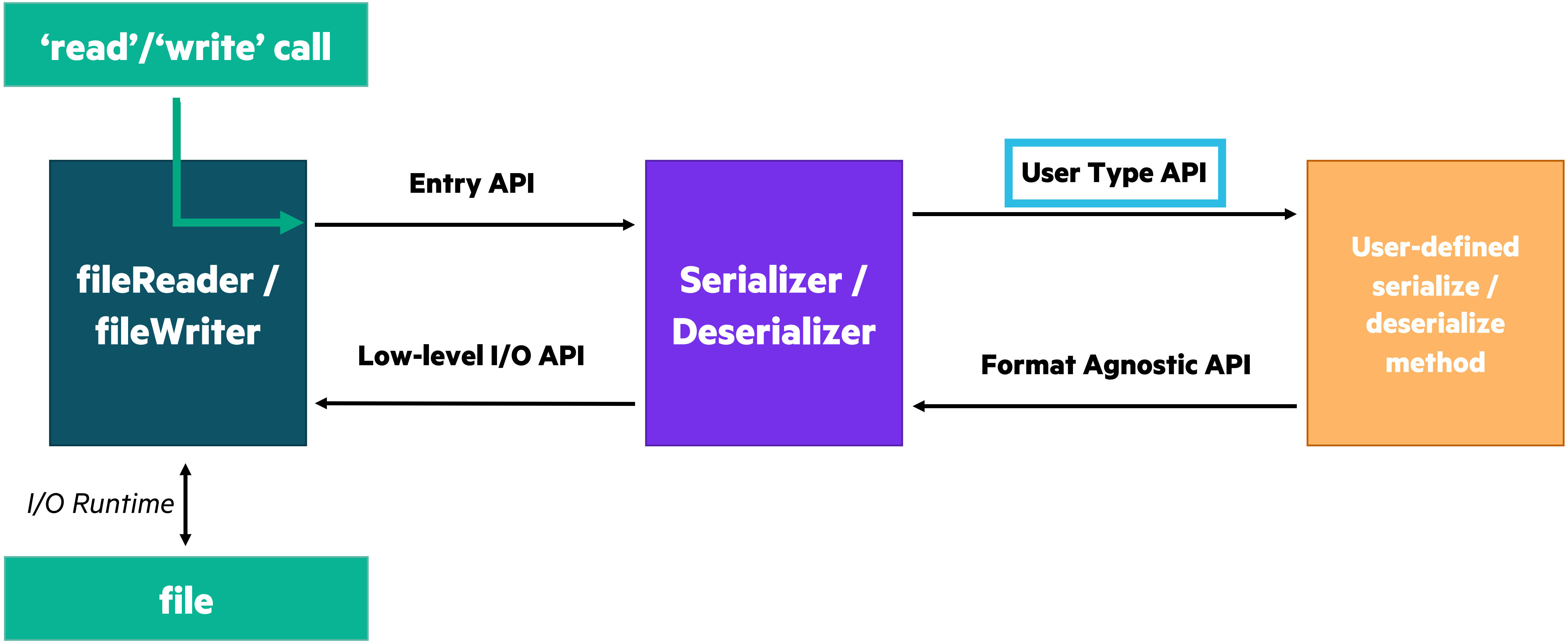
- Deserializers provide six 'start' methods to begin deserializing a kind of type

- Each 'start' method takes a 'fileReader', and returns an object with methods for the specific type-kind
  - The various 'read' methods accept either a value by 'ref', or a 'type', to match 'fileReader.read'

Class	Record	Tuple	Array	List	Map
startClass	startRecord	startTuple	startArray	startList	startMap
readField	readField	readElement	readElement	readElement	readKey
startClass*			startDim		readValue
			endDim	hasMore	hasMore
endClass	endRecord	endTuple	endArray	endList	endMap

* note: second 'startClass' exists to support inheritance

USER TYPE API



USER TYPE API
The 'serialize' Method

- Users may override default serialization behavior with a 'serialize' method
  - The 'serialize' method is defined by the 'writeSerializable' interface:

```
proc T.serialize(writer: fileWriter(?), ref serializer: ?st) throws
```

- Example usage: Write a type as an abstract 'List':

```
// first, explicitly indicate interface
record MyList : writeSerializable { ... }

// Write once, use with any Serializer
proc MyList.serialize(writer: fileWriter(?), ref serializer: ?st) throws {
  var ser = serializer.startList(writer, this.numElements); // in JSON, write "["
  for elem in this do
    ser.writeElement(elem); // in JSON, write "," if necessary, then 'elem'
  ser.endList(); // in JSON, write "]"
}
```

USER TYPE API

The ‘deserialize’ Method

- Users may override default in-place deserialization behavior with a ‘deserialize’ method
  - Intended to provide behavior for ‘fileReader.read’ that accepts values by-ref
  - The ‘deserialize’ method is defined by the ‘readDeserializable’ interface:

```
proc ref T.deserialize(reader: fileReader(?), ref deserializer: ?dt) throws
```

- Example usage: Read a type as an abstract ‘List’:

```
record MyList : readDeserializable { ... }

// Write once, use with any Deserializer
proc ref MyList.deserialize(reader: fileReader(?), ref deserializer: ?dt) throws {
  this.clear(); // reading in-place, so clear the data
  var des = deserializer.startList(reader);
  while des.hasMore() do
    this.add(des.readElement(this.eltType));
  des.endList();
}
```

USER TYPE API

The Deserializing Initializer

- Users may override default 'read(type)' deserialization behavior with an initializer
  - Useful for types that cannot be default-initialized
  - The initializer signature is defined by the ‘initDeserializable’ interface:

```
proc T.init(reader: fileReader(?), ref deserializer: ?dt) throws
```

- Initializer may throw, but only after all fields are initialized
  - Future versions of Chapel may relax this requirement

- Otherwise, works the same as a ‘deserialize’ method

- See IO Serializers technote for information on initializing generic types while deserializing

CUSTOM TYPE SERIALIZATION
Other API Notes

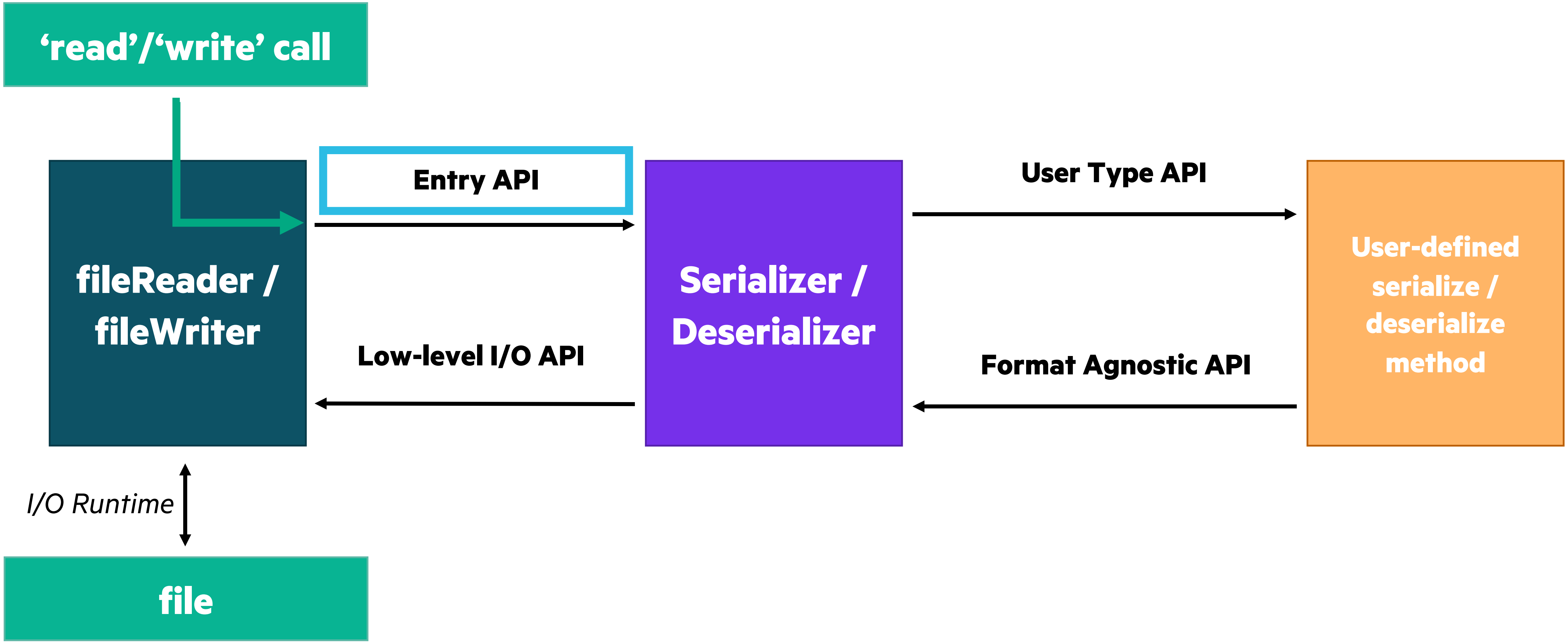
- User types implementing all three methods can use the combined ‘serializable’ interface

- ‘serialize’ and ‘deserialize’ methods on classes must use ‘override’
  - Required because all classes inherit from the RootClass, which can itself be serialized or deserialized

- Implementing ‘serialize’, ‘deserialize’, or an initializer prevents compiler-generation of all three
  - Rationale: User has possibly diverged from default behavior, so do not generate incompatible implementations

IMPLEMENTING (DE)SERIALIZERS

SERIALIZER API DESIGN



IMPLEMENTING SERIALIZERS

The ‘serializeValue’ Method

- To develop a Serializer, users must first implement a ‘serializeValue’ method on a record

```
proc Serializer.serializeValue(writer: fileWriter, const val: ?) throws
```

- ‘serializeValue’ accepts either primitive types, or types with the ‘writeSerializable’ interface

- Once invoked, ‘serializeValue’ has complete control over serialization

- Users must also implement the format-agnostic API of the previous section

IMPLEMENTING DESERIALIZERS

The ‘deserializeValue/Type’ Methods

- To develop a Deserializer, users must first implement ‘deserializeValue’ and ‘deserializeType’ methods

```
proc Deserializer.deserializeType(reader: fileReader,
                                  type readType) : readType throws

proc Deserializer.deserializeValue(reader: fileReader,
                                   ref val: ?readType) : void throws
```

- These methods accept types with either the ‘readDeserializable’ or ‘initDeserializable’ interface
  - Or primitive types

- Once invoked, these methods have complete control over deserialization

- Users must also implement the format-agnostic API of the previous section

STATUS AND NEXT STEPS

I/O SERIALIZERS
Status

- Serializers and Deserializers are available in Chapel 1.32, with several available formats
  - In stable standard libraries: default, binary, JSON formats
  - In unstable package modules: YAML, ChplFormat

- Support for reading and writing in JSON is significantly improved
  - Due to format-agnostic (De)Serializer API

- Users may implement their own (De)Serializers that integrate cleanly with normal use of the ‘IO’ module

I/O SERIALIZERS

Next Steps

- Look for quality-of-life improvements
  - For example, an optional ‘serializer’ argument to ‘writeln’, instead of using ‘withSerializer’ to create an alias

- Provide a more robust binary I/O format
  - Current format is very simplistic
  - Intended to replicate most of the legacy binary I/O behavior provided by ‘iokind’
  - Could improve support for storing redundant class instances

- Explore support for other formats
  - E.g., python’s “pickle”, or converting the TOML package module to use Serializers instead

OTHER IO STABILIZATION CHANGES

FORMATTED IO IMPROVEMENTS

Background and This Effort

Background:

- The 'IO.FormattedIO' module provides C-like IO capabilities such as 'writef' and 'readf'

This Effort:

- Adjusted several format string options
  - left, center, and right justification can be designated with '%<', '%^', and '%>' respectively, e.g.,

```
writef("|%<5i|%^5i|%>5i|", 1, 2, 3);   // writes: "|1    |  2  |    3|"
```

  - made real number formatters respect precision for integer arguments

```
writef("%.5r", 1);  // writes: "1.00000"
```

  - made integer formatters emit a warning for ignored precision arguments

```
writef("%.5i", 1);  // writes: "1" (emits a runtime warning)
```

  - replaced %t, %jt, and %ht with %? and serializers:

```
record r { var x: int; }        stdout.writef("%?", new r(1));   // writes: "(x = 1)"
stdout.withSerializer(jsonSerializer).writef("%?", new r(2));   // writes: "{"x":2}"
stdout.withSerializer(chplSerializer).writef("%?", new r(3));   // writes: "new r(x = 3)"
```

FORMATTED IO IMPROVEMENTS

Impact

- Addresses inconsistency between '%-' for left justification and '%+' for printing a '+' with positive numbers
- Precision specifiers behave more consistently across types
- (De)Serializers can now control the behavior of the "any type" format specifier
  - special formats like JSON are no longer built into the IO runtime

GENERAL IO IMPROVEMENTS

This Effort:

- Updated 'readLiteral' and 'matchLiteral' to respect leading whitespace in the literal string
  - the literal's leading whitespace must match for the literal to match, even for 'ignoreWhitespace=true', e.g.,

```
myFile.reader().matchLiteral("   asdf", ignoreWhitespace=true);
```

- Updated IO runtime to not buffer for sufficiently large read or write operations
- Generalized '[read|write]Binary' to support multi-dimensional arrays

Impact:

- 'readLiteral' and 'matchLiteral' no longer ignore leading whitespace characters in the literal string
- Avoiding buffering can improve performance for programs with large IO operations
  - allowed undocumented 'QIO_CHANNEL_ALWAYS_UNBUFFERED' flag to be removed from some benchmarks
- Improved usability for bulk binary IO with arrays

IO DEPRECATIONS

This Effort:

Deprecated Symbol	Replacement
file[Reader|Writer].writing	type check
file[Reader|Writer].binary	check against (de)serializer type
file[Reader|Writer].kind	using the binary (de)serializer with fileReader/fileWriter
ioLiteral	'fileReader.[read | match]Literal' and 'fileWriter.writeLiteral'
ioNewline	'fileReader.[read | match]Newline and 'fileWriter.writeNewline'
fileReader.readWriteLiteral	'fileWriter.writeLiteral'
fileWriter.readWriteLiteral	'fileReader.readLiteral'
fileReader.readWriteNewline	'fileReader.readNewline'
fileWriter.readWriteNewline	'fileWriter.writeNewline'

Impact:

- Distinguishing 'fileReader's and 'fileWriter's via the type system is encouraged
- Queries on 'fileReader' and 'fileWriter' are replaced with new (de)serializer equivalents
- The interface for reading/writing string literals and newlines is now simplified

MATH/AUTOMATH MODULES

MATH/AUTOMATH MODULES

Background and Actions Taken/Decisions Made

Background:

- Provides mathematical constants and functions, e.g., 'e', 'sqrt( )', 'gcd( )'
- 'AutoMath' is included in all programs by default, 'Math' requires a 'use' or 'import' to access

Actions Taken/Decisions Made:

- Stopped including more symbols by default, e.g., 'e', 'pi', 'erf( )', 'log( )'
- Unified argument names to 'x' and 'y'

Before, for example:

```
inline proc conjg(z: real(?w)) { … }
inline proc log2(val: int(?w)) { … }
proc log1p(x: real(64)): real(64) { … }
proc divceil(m: integral, n: integral) { … }
```

After:

```
inline proc conjg(x: real(?w)) { … }
inline proc log2(x: int(?w)) { … }
proc log1p(x: real(64)): real(64) { … }
proc divceil(x: integral, y: integral) { … }
```

MATH/AUTOMATH MODULES

Actions Taken/Decisions Made, and Next Steps

Actions Taken/Decisions Made (continued)**:**

- Renamed many functions for clarity and to align with our standard module style guidelines
  - e.g., renamed 'carg( )' to 'phase( )' and 'cproj( )' to 'riemProj( )'
- Marked several symbols as unstable for 2.0
  - including 'nearbyint( )' and 'erf( )’
- Marked the ‘AutoMath’ module name as unstable, reflecting a vision of its contents being part of ‘Math’
  - Enabled ‘AutoMath’ symbols to use 'Math.' for qualified access, e.g.

```
writeln(Math.cbrt(27));    // 'cbrt( )' is available by default via the 'AutoMath' module but can use 'Math.' as a prefix
```

Next Steps:

- Stabilize remaining symbols
- Implement more extensive rounding support
- Fold the documentation for ‘AutoMath’ into the ‘Math’ module documentation itself

BIGINTEGER MODULE

BIGINTEGER
Background and This Effort

Background: The ‘BigInteger’ module provides a Chapel-tastic multiple precision integer type, ‘bigint’

This Effort:

- Converted overwriting methods to free functions

```
var result, x, y: bigint;
x = 5: bigint;
y = 12: bigint;
add(result, x, y); // used to be 'result.add(x, y)'
```

- Unified procedure names to the Chapel style
  - Consistent casing, e.g., ‘addmul( )’ to ‘addMul( )’
  - Improved clarity, e.g., ‘divQ( )’ to ‘div( )’
- Unified argument names to a consistent naming scheme
  - Most procedures take arguments named ‘x’ and ‘y’
  - Some arguments denote special meaning, e.g., ‘result’, ‘n’, and ‘exp’
- Renamed ‘round’ enum to ‘roundingMode’

BIGINTEGER

This Effort and Status

This Effort (continued)**:**

- Added cast from ‘bool’

```
var x = true: bigint;
```

- Deprecated ‘get_str’ in favor of casting to a string

```
var myStr = new bigint(17): string;
```

- Improved performance with remote-value-forwarding for ‘bigint’
- Marked infrequently used procedures we aren’t sure about as unstable (e.g., ‘legendre( )’)
- Deprecated the transitional ‘config param bigintInitThrows’
- Removed previously deprecated symbols (e.g., ‘fits_*( )’)
- Refreshed documentation and refactored code
- We considered renaming the module to ‘BigInt’ to match the type ‘bigint’, but did not go forward with it

Status: The ‘BigInteger’ module is now stable

COLLECTION TYPES

COLLECTION TYPES

This Effort:

- Renamed some 'list' methods
  - 'push' -> 'pushBack'
  - 'pop' -> 'popBack' / 'getAndRemove'
  - 'set' -> 'replace'
- Renamed 'map.addOrSet' to 'map.addOrReplace'
- Removed some limitations with 'map'
  - indexing with a default-initializable value no longer throws
  - 'map.values( )' is available for maps with non-nilable owned values
- Marked 'parSafe' fields on 'list', 'map' and 'set' unstable
- Marked 'list.sort' unstable

Impact:

- 'list' and 'map' method names more clearly reflect their behavior
- Improved 'map's usability across a wider variety of types
- The unstable warning for 'parSafe' indicates intention to add separate parallel-safe types in the future

ERRORS MODULE

ERRORS MODULE

Background:

- The 'Errors' module contains the base 'Error' class and other standard error types

This Effort:

- Renamed 'codepointSplittingError' to 'codepointSplitError'
- Deprecated the two-argument initializer for 'IllegalArgumentError'

Impact:

- Improved consistency in tense of error names
- Unified initializer signatures across error types

COLLECTIVES MODULE

COLLECTIVES MODULE

This Effort:

- Deprecated non-reusable barriers and the initializer argument for requesting them

```
use Collectives;

// warning: non-reusable barriers are deprecated, please remove the 'reusable' argument from this initializer call
var b = new barrier(4, reusable=true);
```

- Deprecated and renamed the barrier check method

```
use Collectives;
var b = new barrier(4);
if b.check() then        // warning: 'barrier.check()' is deprecated, please use '!barrier.pending()' instead
  …

if !b.pending() then     // use this method instead
  …
```

TIME MODULE

TIME MODULE
Background and This Effort

Background: The 'Time' module provides types for working with dates and times, and time measurement

- Previously reviewed, but not completely stabilized

This Effort: Final re-review of Time module for internal consistency and alignment with current standards

- Deprecated procedures with redundant functionality:
  - 'date'-forwarding 'dateTime' methods 'isoCalendar', 'toOrdinal', 'weekday', 'isoWeekday'
  - 'getCurrentDate', 'getCurrentDayOfWeek', 'MINYEAR'/ 'MAXYEAR' in favor of 'date' type methods
  - 'date.createFromTimestamp', in favor of 'dateTime' method
  - 'isoFormat' methods, in favor of string cast or other formatting methods
  - 'dateTime.combine(date, time)', in favor of corresponding 'init'
- Pared down day-of-week enums to just one 'dayOfWeek' matching previous 'isoDayOfWeek'
- Fixed asymmetrical behavior w.r.t. UTC and local versions of current-time methods, improved documentation
- Marked 'Timezone' and all procedures using it as unstable
- Renamed symbols inconsistent with our naming and casing conventions

TIME MODULE

This Effort, Impact, and Next Steps

This Effort (continued)**:**

- Made several documentation improvements, including explicit return types on all procedures
- Renamed ‘isoCalendar’ to ‘isoWeekDate’
- Converted free function ‘abs(timeDelta)’ to method ‘timeDelta.abs( )’

Impact:

- Improved module consistency and clarity of documentation
- Reduced ways to get the same information (net ~15 symbols deprecated)

Next Steps:

- Implement monotonic timers
- Make timezone awareness/naïveté part of ‘dateTime’ and ‘time’ static types
- Consider supporting timing via attributes or context managers, in addition to manual ‘stopwatch’ use
- Support ‘%f’ format specifier in ‘dateTime.strptime’

FILESYSTEM

FILESYSTEM

Background:

- The 'FileSystem' module focuses on file and directory properties and operations
- 'umask' sets the file permissions that all new files will inherit
- We have not decided how 'umask' should behave on non-CPU locales (i.e., GPUs)

This Effort: Marked 'umask' as unstable on locale models other than 'flat'

Next Steps: Determine how 'umask' should behave in other locale models

REFLECTION

REFLECTION

Background: The 'Reflection' module offers support for reflecting about properties of Chapel code

This Effort:

- Deprecated 'fieldName' in favor of 'getFieldName'
- Marked several procedures unstable:
  - 'isFieldBound': Check if a type's field is instantiated, consider using 'T.fieldName != ?' syntax instead
  - 'canResolve…': Check to see if a call resolves
  - 'getFieldRef': Get a mutable reference to an instance field

Next Steps: Add stable replacements for some unstable features

- Combining 'getField' with 'getFieldRef' may require changes to the language
- Add a 'canResolve' procedure to check if expressions resolve

CTYPES MODULE

CTYPES MODULE
Background and This Effort

Background: ‘CTypes’ provides Chapel representations of C types, supporting interoperability procedures

This Effort: Improved ‘c_ptr’ and distilled functionality to focus on C interoperability
- Made ‘c_ptr’ and ‘c_ptrTo’ improvements — see 'c_ptr' improvements slides for more information
- Combined ‘c_malloc’/’c_calloc’/’c_aligned_alloc’/’c_free’ procedures into new ‘allocate’/’deallocate’ interface:

```
proc allocate(type eltType, size, clear = false, alignment = 0): c_ptr(eltType)
proc deallocate(data: c_ptr(void))
```

- Included unstable ‘strLen’ and ‘c_str’ functions to support ‘c_string’ replacement with ‘c_ptr’s
  - See ‘c_string’ slides for more information

CTYPES MODULE
This Effort, Impact and Next Steps

This Effort (continued)**:**

- Moved 'c_mem{move,cpy,cmp,set}' into 'OS.POSIX' without 'c_' prefixes, with consistent formal types
- Deprecated 'c_nil', 'is_c_nil', and 'isAnyCPtr'
- Deprecated cast from class types to 'c_ptr(void)' in favor of 'c_ptrTo'
- Made documentation improvements, including in "C Interoperability" technote

Impact:

- C pointers can be used with more types, and support more useful situations
- Module functionality is more specifically focused on C interoperability

Next Steps:

- Provide coherent external array interoperability between CTypes facilities and 'chpl_external_array' [#16135]

CHPLCONFIG MODULE

CHPLCONFIG MODULE

Background: The ‘ChplConfig’ module provides compile-time Chapel configuration information

- Contains many ‘CHPL_*’ param strings: ‘CHPL_HOME’, ’CHPL_TARGET_COMPILER’, ’CHPL_COMM’, …

This Effort: Began moving away from ‘CHPL_*’ variables in favor of user-facing query procedures

- Added a ‘compiledForSingleLocale( )’ query
  - Motivated by frequent checks for whether ‘CHPL_COMM == none’
  - Result determined by ‘--[no-]local’ flag if present, or ‘CHPL_COMM’ variable otherwise
- Marked all ‘CHPL_*’ variables unstable

Next Steps: Continue the transition towards nice user-facing queries for config information

- Add more useful queries for checking ‘CHPL_*’ variable information
- Remove ‘CHPL_*’ variables as they become unneeded

BITOPS

BITOPS MODULE

Background:

- The ‘BitOps’ module contains utilities for bit manipulation

This Effort:

- Renamed ‘popcount( )’ to ‘popCount( )’

Status:

- The ‘BitOps’ module is now stable

LIBRARY STABILIZATION: NEXT STEPS

CHAPEL 2.0 LIBRARY STABILIZATION

Next Steps

- Stabilize remaining unstable symbols in vetted modules
  - e.g., 'BigInteger.gcd( )', 'Reflection.canResolve( )'

- Stabilize remaining standard modules
  - e.g., CommDiagnostics, GMP, Help, GPU, Random, Heap

- Stabilize package modules and remaining distributions
  - e.g., ZMQ, LinearAlgebra, ArgumentParser

- Use stabilization process when designing new features
  - Features will still be prototypical, but should reduce the chance of subsequent renamings

CHAPEL 2.0 LIBRARY STABILIZATION

Next Steps

- Document '@deprecated' and '@unstable' attributes as user-facing features
  - Developers can use them when making changes

- Implement parallel and distributed versions of Map, Set, and List using their stabilized interface

- Reduce uses of unstable features in release examples directory

OTHER LIBRARY IMPROVEMENTS

OTHER LIBRARY IMPROVEMENTS

For a more complete list of library changes and improvements in the 1.31 and 1.32 releases, refer to the following sections in the CHANGES.md file:

- Namespace Changes
- Standard Library Modules
- Package Modules
- Standard Domain Maps (Layouts and Distributions)
- Changes/Feature Improvements in Libraries
- Name Changes in Libraries
- Name Changes in the 'Math' Library
- Name Changes in the 'BigInteger' Library
- Other Name Changes in Libraries
- Deprecated/Unstable/Removed 'IO' Library Features
- Deprecated/Unstable/Removed 'Math' Library Features
- Deprecated/Unstable/Removed 'Time' Library Features
- Unstable Library Features
- Deprecated/Removed Library Features
- Deprecated/Unstable/Removed Library Features
- Performance Optimizations/Improvements
- Documentation Improvements for the 'IO' Library
- Documentation Improvements for the 'Math' Library
- Other Documentation Improvements
- Error Messages/Semantic Checks
- Bug Fixes for Libraries
- Developer-oriented changes: Module changes

THANK YOU

https://chapel-lang.org
@ChapelLanguage