Hewlett Packard
Enterprise

CHAPEL 1.29.0/1.30.0 RELEASE NOTES: LIBRARY IMPROVEMENTS

Chapel Team

December 15, 2022 / March 23, 2023

OUTLINE

- Weak Pointers
- 'BigInt' Improvements
- Chapel 2.0 Stabilization
- Other Library Improvements

WEAK POINTERS

WEAK POINTERS
Background

- 'shared' memory management allows multiple variables to refer to the same class instance
  - When the last 'shared' variable pointing to a class is deinitialized, the class's memory can be freed
  - This is accomplished in a parallel-safe manner using atomic reference counting

```
{
  var s1 = new shared C();   // reference count: 1
  {
    var s2 = s1;             // reference count: 2
  }                          // reference count: 1
  var s3 = s1;               // reference count: 2
}                            // reference count: 0
```

- Some other languages and libraries supporting similar functionality pair it with a *weak pointer* type
  - A weak pointer refers to some 'shared' variable, but doesn't require it to stay allocated
  - This can be useful for controlling deallocation in a variety of situations:
    - in the presence of cyclical references
    - maintaining a cache of references to objects

WEAK POINTERS
This Effort

- Added an experimental 'weak' type to the standard library
  - The interface design is based heavily on Rust's 'Weak' type

- Weak pointers are meant to be used in tandem with 'shared' classes
  - Holding a 'weak' reference to a 'shared' class does not prevent it from being deallocated
    - I.e., the behavior of 'shared' itself is not affected by this change
  - A 'weak' reference must be *upgraded* into a 'shared' class before it can be used as a class variable
  - If the referenced 'shared' has already been deallocated, i.e., its reference count is zero, upgrading will fail
    - If upgrading into a nilable type, the result will be 'nil'; otherwise, an error will be thrown

WEAK POINTERS
Supported Conversions

- 'weak' supports a few options for converting to/from 'shared'

shared -> weak

- 'downgrade' method:

```
var myC = new shared C(),
    weakC = myC.downgrade();
```

- weak initializer:

```
var myC = new shared C(),
    weakC = new weak(myC);
```

weak -> shared

- 'upgrade' method:

```
var maybeC = weakC.upgrade();
if maybeC != nil { ... }
```

- cast to a nilable shared:

```
var maybeC = weakC: shared C?;
if maybeC != nil { ... }
```

- cast to non-nilable shared:

```
try {
  var c = weakC: shared C;
  ...
} catch e: NilClassError {
  ...
}
```

WEAK POINTERS

Impact: weak cache example

- It is now possible to implement data structures like a “weak cache” that:
  - maintain a set of 'shared’ classes, but do not force them to stay allocated
  - upon request, retrieves the ‘shared’ class if it is still allocated, otherwise constructs a new one using a ‘builder’ function

```
use WeakPointer, Map;
record weakCache {
  type t;                                    // cached 'shared' class type
  var items: map(string, weak(t));           // map of weak ptrs
  proc getOrBuildShared(key: string, builder): t {
    if items.contains(key) {                 // have a 'weak' ptr for this key?
      var s : t? = items[key].upgrade();
      return if s != nil                     // found a shared class?
        then s: t                            // yes: cast away nilability
        else saveWeak(key, builder(key));    // no: make new one
    } else {
      return saveWeak(key, builder(key));    // no: make new one
    }
  }
  proc saveWeak(key: string, s: t): t {
    items[key] = s.downgrade();
    return s;
  }
}
```

WEAK POINTERS

Impact: weak cache example (continued)

```
class C { var x: string; }

// define a builder function (using new FCF syntax)
const builder = proc(k: string) {
  writeln("building: ", k);
  return new shared C(k);
};

{
  // create a 'weakCache' using the type defined on the previous slide
  var wc = new weakCache(shared C);
  {
    var s1 = wc.getOrBuildShared("A", builder);
    var s2 = wc.getOrBuildShared("A", builder);
  }
  var s3 = wc.getOrBuildShared("A", builder);
}
```

```
// the following are the counts for key ”A" in the weak cache:
// initially, the cache doesn’t hold “A” so there are no counts
// shared count: 1, weak count: 1,  writes: "building: A"
// shared count: 2, weak count: 1
// shared count: 0, weak count: 1  (s1 & s2 deallocated)
// shared count: 1, weak count: 1,  writes: "building: A"
// shared count: 0, weak count: 0  (cache is deallocated)
```

WEAK POINTERS
Status and Next Steps

Status

- 'weak' is still in its experimental stage, and is marked as unstable

Next Steps

- Resolve some open interface questions:
  - Which of the "downgrade" paths (cast, method, & initializer) should be supported? [#20949]
  - How to access the corresponding 'shared' type? [#20952]
  - Which operators and special methods should be supported? [#20951]
- Decide on a module name and location [#20956]
  - Should ‘weak’ by part of the language? Should ‘shared’ be part of the standard library?
  - If both ‘weak’ and ‘shared’ are both defined in a standard module, should it be auto-use’d?
- Implement final design and mark as stable

‘BIGINT’ IMPROVEMENTS

‘BIGINT’ IMPROVEMENTS

Background

- The Chapel ‘bigint’ type is a record that wraps GMP’s multiple precision integer
  - Stores limbs, sign, magnitude, and other information as a field of the external C ‘mpz_t’ type
- Handles multi-locale execution, arithmetic operator overloads, and automatic memory management
- Recent inclusion of the ‘bigint’ type in Arkouda led to greater scrutiny of the module
- When creating a ‘bigint’, the ‘mpz_t’ buffer is created on the current locale
  - In distributed settings, execution is often migrated to the locale owning the buffer, to pass it to extern C routines
  - When operating on multiple ‘bigint’s, execution is performed on the LHS locale and the RHS is localized
    - i.e., a local copy is made if it isn’t already local

‘BIGINT’ IMPROVEMENTS

This Effort and Impact

This Effort:

- Refactored ‘BigInteger’ module, resulting in less code duplication and greater clarity
- During this refactor, several bugs were caught, exposing gaps in the existing ‘bigint’ testing
  - Lacked tests of remote ‘bigint’ values
  - Lacked tests of ‘bigint’ values larger than 64 bits
  - Lacked tests comparing results of 64-bit ‘bigint’ values against Chapel integers
- Added testing of full ‘bigint’ API with remote/massive values and comparisons against Chapel integers

Impact:

- Found and fixed 6 ‘bigint’ correctness bugs
- Removed about 600 lines from the ‘BigInteger’ module
- Reduced code duplication, simplifying code maintenance
- Has the potential to reduce compilation times for ‘bigint’-heavy codes
- Added a fraction of a millisecond overhead to affected ‘bigint’ functions

‘BIGINT’ IMPROVEMENTS

Next Steps

- Plan to implement ‘serialize’/‘deserialize’ methods for ‘bigint’, enabling *remote value forwarding*
  - An optimization that transfers values with the message bundle used to implement an ‘on’ statement
  - Helps reduce overhead by eliminating remote reads that would otherwise be needed to fetch read-only data
- Continue to explore opportunities for code simplification
- Continue improving and stabilizing ‘bigint’ methods and routines

CHAPEL 2.0
LIBRARY STABILIZATION

CHAPEL 2.0 LIBRARY STABILIZATION

Background and Status

Background:

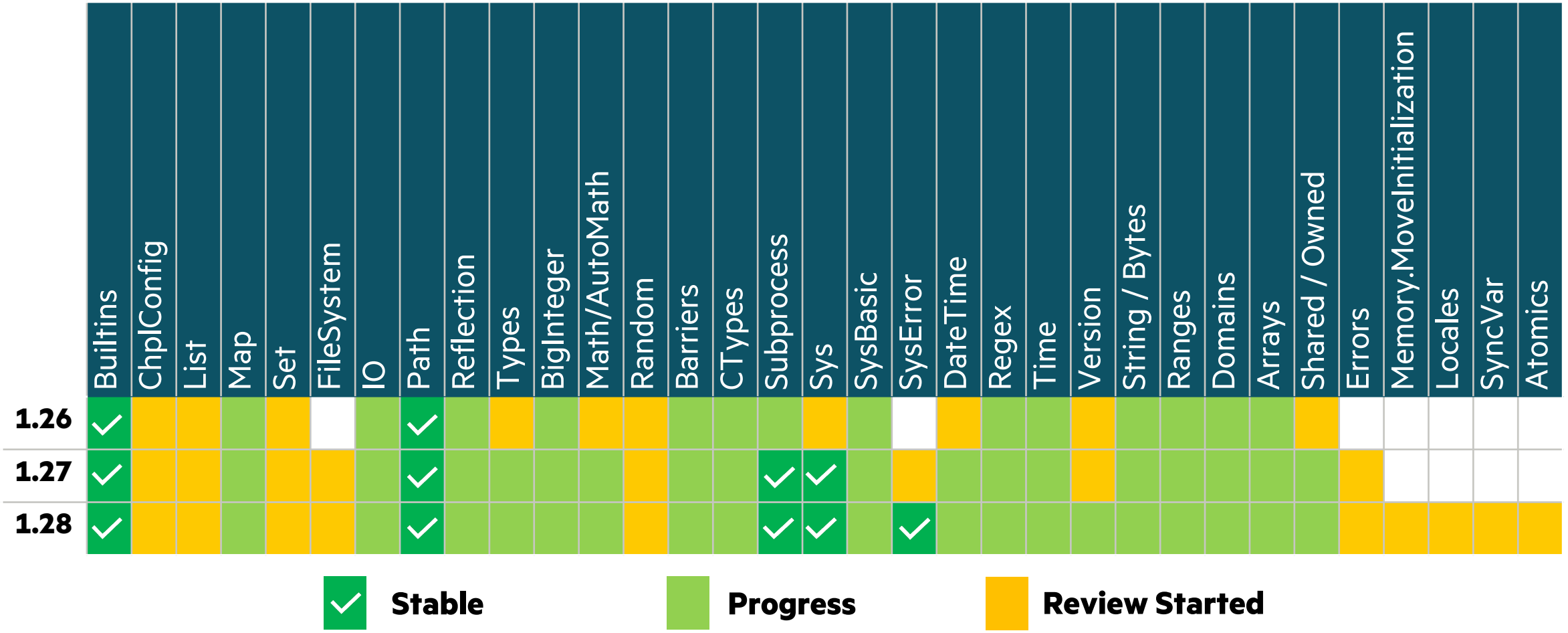
- Our primary focus is standard library stabilization
  - *Stabilization:* Going forward, all changes will be backwards-compatible
  - Users should be able to depend on anything not marked '@unstable' to continue working through all 2.X releases.

Status In Numbers:

- 38 modules reviewed
- 12 modules stabilized:
  - Path, Builtins, Subprocess, SysError, Sys, Locales, Types, SysBasic, Regex, Version, Arrays, MemMove
- 9 modules estimated for 1.31:
  - CTypes, Time, DateTime, FileSystem, String/Bytes , Map, List, Errors
- 14 modules estimated for 1.32:
  - BigInteger, Math, IO, Collectives, Set, ChplConfig, Ranges, Owned/Shared, Domains, Reflection, Sync/Single/Atomics
- 10 modules that we've decided not to stabilize before Chapel 2.0:
  - CommDiagnostics, Memory[.Diagnostics], BitOps, GMP, DynamicIters, VectorizingIterator, Help, GPU, GpuDiagnostics, Random

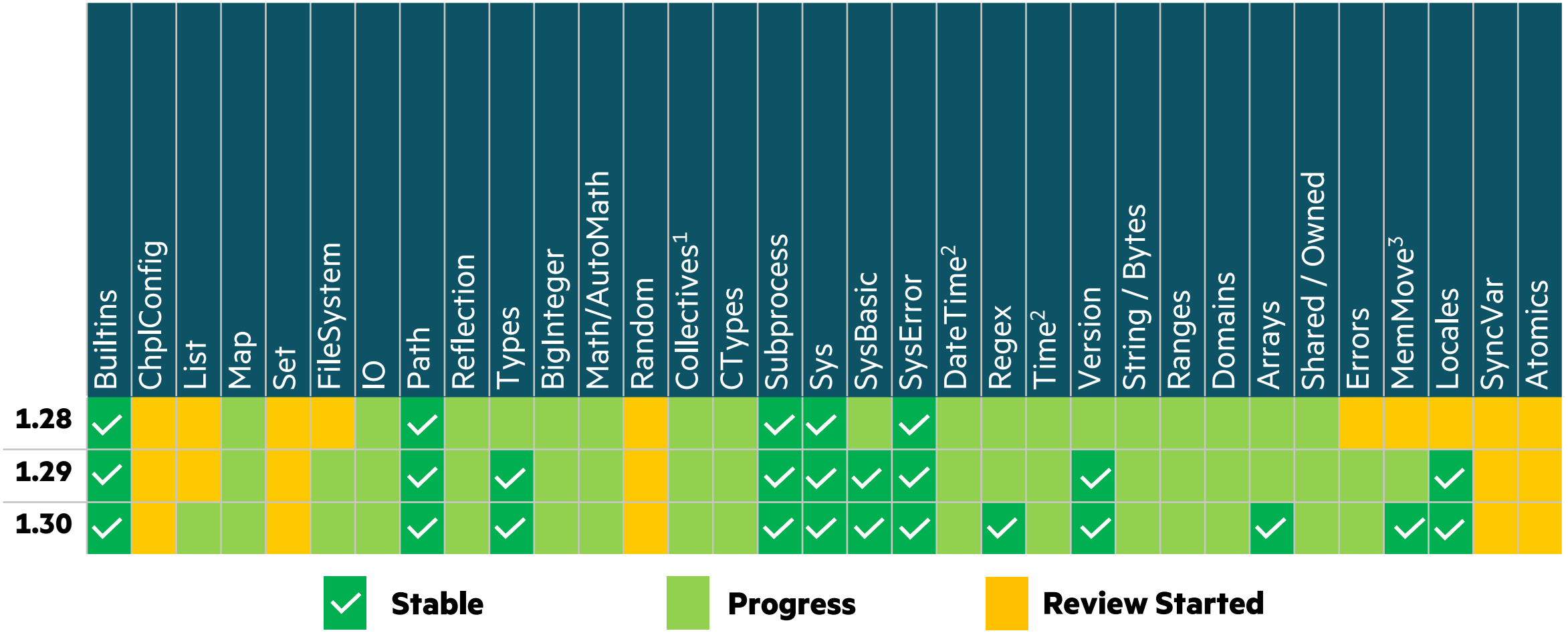
CHAPEL 2.0 LIBRARY STABILIZATION

Status: Visualized



CHAPEL 2.0 LIBRARY STABILIZATION

Status: Visualized



[1] — Barriers was renamed to Collectives

[2] — DateTime and Time were combined into a single module

[3] — Memory.MoveInitialization was renamed to MemMove

LIBRARY STABILIZATION OUTLINE

- IO
- Collectives
- Distribution Modules
- Errors
- FileSystem
- MemMove
- Regex
- SysBasic
- Time
- Types
- Version

IO MODULE
Background

- The 'IO' module handles reading and writing to files, as well as formatted IO
  - ‘write( )’, ‘writeln( )’ and ‘writef( )’ are provided by default, all other IO functions are defined in the 'IO' module
- Contains 'file' and 'channel' types
- This module is very large, ~7300 lines

IO MODULE

This Effort

- Split 'channel' type into 'fileReader' and 'fileWriter'
- Developed prototype Serializer/Deserializer mechanism
  - Both for supporting default reading/writing behavior and reading/writing in JSON format
- Added new methods 'readAll( )', 'readThrough( )', and 'readTo( )'
- Added new overloads for 'readBinary( )' and 'writeBinary( )'
- Made 'region' arguments inclusive of their bounds
- Made 'file.path' exclusively return absolute paths
- Removed unnecessary 'bool' return values from 'write' functions
- Unified 'ioHintSet.mmap' and '.noMmap' into a single type method, 'ioHintSet.mmap(useMmap: bool)'
- Deprecated 'file.localesForRegion( )' and 'unicodeSupported( )'
  - Unicode is always supported
- Marked 'iostringstyle' and 'iostringformat' as unstable
- Renamed or replaced an additional 9 routines, methods and types

IO MODULE
Next Steps

- Implement resolved design decisions:
  - Add 'stripNewline' argument to 'fileReader.lines( )'
  - Replace 'fileReader/fileWriter.binary( )' with new binary serializer/deserializer
  - Deprecate '%j' and '%h' format string specifiers in favor of serializers/deserializers
  - Unify methods like 'commit( )' and '_commit( )' into a single method and document behavior w.r.t. locking
- Implement other serializers/deserializers (e.g., binary, "Chapel format", YAML)
- Resolve open decisions
  - How should '%t' behave w.r.t. serializers/deserializers? [#19906]
  - Should 'assertEOF( )' be deprecated? [#19316]
  - What should be done with the 'iokind' field on 'fileReader'/'fileWriter'? [#19314]
  - Should the 'writing' method remain on 'fileReader'/'fileWriter' or be deprecated?

COLLECTIVES MODULE
Background

- The ‘Barriers’ module has supported a ‘Barrier’ record type
  - Provides a task barrier with two implementations:
    - One that uses atomics, the other ‘sync’ variables
  - User could select between them when creating new instances of ‘Barrier’

```
var b = new Barrier(numTasks, BarrierType.Sync);
```

    - If unspecified, ‘Atomic’ was the default
    - Implementation used dynamic dispatch to switch between the two versions
  - The ‘Sync’ version was not typically used in practice

COLLECTIVES MODULE

This Effort and Next Steps

This Effort:

- Decided to only support the 'Atomic' implementation going forward
  - Will remove the need for dynamic dispatch on each call once the deprecated 'Sync' implementation is removed
- Renamed 'Barrier' to 'barrier' to match the naming convention for records
  - Removed an outdated compiler error about methods whose names matched their type
    - This error was introduced when initializers replaced constructors in Chapel

```
proc barrier.barrier() …   // is now allowed!
```

- Renamed the 'Barriers' module to 'Collectives'
  - There is only one 'barrier' type, and we expect other collectives to be added over time
- Deprecated the 'BarrierType' enum

Next Steps:

- Remove the 'BarrierType' enum and the dynamic dispatch-based implementation
  - Should improve the speed of barrier method calls significantly

DISTRIBUTION MODULES: BLOCKDIST AND CYCLICDIST

Background:

- 'BlockDist' and 'CyclicDist' are used to partition a domain's indices / array's elements across locales
- These modules have supported standalone factory routines to generate new domains/arrays

This Effort:

- Renamed the factory routines and made them into type methods
  - New names are more consistent with factory routine naming in other modules:
    - 'newBlockDom(...)' is now 'Block.createDomain(...)'
    - 'newBlockArr(...)' is now 'Block.createArray(...)'
    - 'newCyclicDom(...)' is now 'Cyclic.createDomain(...)'
    - 'newCyclicArr(...)' is now 'Cyclic.createArray(...)'

Impact:

- New routines are clearer, better organized, and support generic programming across distributions

```
const D = myDist.createDomain(1..n);
```

ERRORS MODULE

Background: The 'Errors' module contains common error types and related routines

- Provides the base class 'Error' and some of its child classes
- Provides error and halting procedures, such as: 'assert( )', 'compilerError( )', 'exit( )', 'halt( )', etc.

This Effort: Minor consistency and naming improvements

- Unified varargs formatting by removing queries for the number of arguments from all procedures in the module
  - Only affects documentation
  - Example:

- Renamed argument in 'IllegalArgumentError' initializer from 'info' to 'msg'
  - This matches the formal name in the base 'Error' class:

```
throw new IllegalArgumentError(msg="cannot divide by zero");
```

FILESYSTEM MODULE

Background:

- The ‘FileSystem’ module contains utilities for manipulating files and directories

This Effort:

- Renamed routines to match camelCasing naming conventions:
  – listDir( ), walkDirs( ), getUid( ), getGid( )
- Deprecated the ‘copyFile( )’ routine in favor of ‘copy( )’
- Deprecated the ‘sameFile( )’ overload that takes a ‘file’ argument
  – This was the only routine to take a ‘file’ rather than a path string
- Added an optional ‘metadata’ argument to ‘copyTree( )’

Status:

- A few more minor changes are needed for 2.0 stabilization

MEMMOVE
Background and This Effort

Background: 'Memory.Initialization' module provided move-initialize semantics, but was not ready for 2.0

- Uncommon module structure: no other standard modules are sub-modules
- Procedures in the module required naming improvements

This Effort: Stabilized for 2.0 and renamed as top-level 'MemMove' module

- Added new routines to replace old, deprecated versions

```
proc needsDestroy(type t) param : bool   // replaces 'needsDeinit()'
proc destroy(ref obj: ?t)                // replaces 'explicitDeinit()'
proc moveFrom(const ref src: ?t): t      // replaces 'moveToValue()'
```

- Renamed formals of some routines

```
proc moveInitialize(ref dst, in src)    // formerly 'lhs' and 'rhs'
proc moveSwap(ref x: ?t, ref y: t)       // formerly 'lhs' and 'rhs'
```

MEMMOVE

This Effort (continued) and Status

This Effort (continued):

- Replaced ‘moveInitializeArrayElements( )’ with unstable ‘moveArrayElements( )’
  - Old interface was not idiomatic Chapel and unsuitable for 2.0
  - Need more experience with ‘moveArrayElements( )’ before considering it part of 2.0

```
proc moveArrayElements(ref dst:[] ?eltType, const ref src:[] eltType) : void throws

// a variant to avoid array slicing
proc moveArrayElements(ref dst:[] ?eltType, const dstRegion,
                       const ref src:[] eltType, const srcRegion) : void throws
```

Status: ‘MemMove’ is ready for 2.0

REGEX MODULE

Background: The ‘Regex’ module (formerly ‘Regexp’) was originally based on Python's 're' module

- 'compile( )' was the way to create a 'regex' object from a string

```
var re = Regex.compile("foo");    // 're' is a 'regex' object
```

- ‘sub( )' and 'subn( )' were used for substring replacement based on regex

```
re.sub(myString, replString);     // return a new string where matches of 're' in 'myString' are replaced with 'replString'
re.subn(myBytes, replBytes);      // similar, but return a tuple that has the resulting bytes and number of replacements
```

This Effort: Found parts of the ‘Regex’ interface inconsistent with the standard library

- Deprecated 'compile( )' in favor of 'new regex( )', now that throwing initializers are supported

```
var re = new regex("foo");        // with 1.30, 'regex' initializer should be used
```

- Deprecated ‘sub( )’/'subn( )' in favor of 'replace( )’/'replaceAndCount( )' tertiary methods on ‘string’ and ‘bytes’

```
myString.replace(re, replString);          // similar interface to existing 'string.replace(string)', but in 'Regex' module
myBytes.replaceAndCount(re, replBytes);    // returns a tuple whose second element is the number of replacements
```

Status: ‘Regex’ is now stabilized

SYSBASIC MODULE

Background: Functionality we wanted to preserve had been moved out of the 'SysBasic' module over time

- As of 1.28, contained mostly unused and untested symbols, such as non-POSIX error codes

This Effort: Deprecated entire 'SysBasic' module

- Moved Chapel-specific 'EEOF', 'ESHORT', and 'EFORMAT' error codes to 'OS' and hid from users

Impact: Some unused symbols were deprecated without replacement, reducing maintenance burden

- 'fd_t' alias for 'c_int' for file descriptors
- 'ENOERR' constant with value of 0, which was Chapel-specific
- Linux-specific (non-POSIX) error codes
- Optional/extension POSIX error codes

Next Steps: Removal of 'SysBasic' code in 1.31

TIME MODULE

Background and This Effort

Background:

- The ‘Time’ module provides procedures and types for measuring and reasoning about time

This Effort:

- Renamed the ‘Timer’ type to ‘stopwatch’
  - Added ‘stopwatch’ methods ‘restart( )’ and ‘reset( )’
- Renamed several symbols to match camelCase naming conventions
- Deprecated ‘getCurrentTime( )’ in favor of ‘timeSinceEpoch( ).totalSeconds( )’
- Deprecated the ‘TimeUnits’ type in favor of always using seconds
  - It was only providing the illusion of increased accuracy
  - A more accurate timer can be added as a non-breaking change in the future

TIME MODULE

Status and Next Steps

Status:

- ‘Time’ module is nearly 2.0-ready
- Reached consensus on nearly all symbol names and APIs
- Implemented all approved stabilization changes

Next Steps:

- Reach consensus about ‘datetime’ factory functions
  - Implement any naming changes they require
- Rename a few additional symbols for camelCasing conventions:
  - ‘dateTime’, ‘timeDelta’, ‘day’, ‘getDate’, ‘getTime’
- Implement a monotonic clock and use it where appropriate

TYPES MODULE

Background:

- The 'Types' module contains routines to query and modify types

This Effort:

- Deprecated type/subtype comparison operators in favor of equivalent named procedures
- Removed deprecated 'isFloatType( )'/ 'isFloatValue( )' / 'isFloat( )' functions
  - Previously deprecated due to confusion with 'isReal( )' and behavior of returning 'true' for 'imag' but not 'complex'
  - Use 'isReal( )', 'isImag( )', and/or 'isComplex( )' instead

Status:

- The 'Types' module is now stable

VERSION MODULE

Background: The 'Version' module supports reasoning about version numbers

- For both the 'chpl' compiler and Chapel programs
- To date, it has only supported version values known at compile-time

This Effort:

- Renamed 'sourceVersion' to 'versionValue' to more clearly distinguish compile-time cases
  - Deprecated 'createVersion( )' and recommend using 'new versionValue( )' instead
- Added a 'version' type for working with version numbers at execution time

```
// compile-time example—capable of being used in 'param' conditionals
const verVal = new versionValue(1,30,0);        // 'versionValue' object with values known at compile-time
// execution-time example:
var major, minor, patch : int;
…                                               // assign or adjust values for major, minor, and patch
var ver = new version(major, minor, patch);     // 'version' object with values not known until execution-time
```

Status: Implemented in 1.29.0

Impact: programs can use the new 'version' type to build and reason about version numbers at run-time

OTHER LIBRARY IMPROVEMENTS

OTHER LIBRARY IMPROVEMENTS

For a more complete list of library changes and improvements in the 1.29.0 and 1.30.0 releases, refer to the following sections in the CHANGES.md file:

- 'Standard Library Modules'
- 'Package Modules'
- 'Changes / Feature Improvements in Libraries'
- 'Name Changes in Libraries'
- 'Deprecated / Unstable / Removed Library Features'
- 'Performance Optimizations / Improvements'
- 'Memory Improvements'
- 'Documentation' and 'Other Documentation Improvements'
- 'Bug Fixes for Libraries'

THANK YOU

https://chapel-lang.org
@ChapelLanguage