# Language Improvements

**Chapel Team, Cray Inc.**

**Chapel version 1.13**

**April 7, 2016**

# Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

# Outline

- **Core Language Improvements**
  - Improved Support for Strings
  - Record Memory Management Improvements
  - Use Statement Improvements (‘`use`’)
  - Improving Reduce Intent Support
  - Ref Return Intent Changes
  - Passing Synchronization Variables to Generics
  - Simple Synchronization Types Only
- **Improvements to Built-in Routines**
  - Casting Types to Strings
  - Improved min/max Type Signatures
  - Comparison Operators on int/uint Pairs
  - locale.numPUs()
- **Other Language Improvements**

# Improved Support for Strings

# Strings: Setting

## Background:

- Since v1.11, we've been working on implementing strings as records
  …to remove special-case code from the compiler
  …to serve as a proxy for other interesting value types a user might write
  …as a means of closing string-based memory leaks
- Early drafts ran into problems in our implementation of records
  - Imbalanced constructor/destructor calls, memory leaks, …
  - Early record design/implementation was overly focused on POD cases
  - Strings serve as an acid test for record semantics given their pervasive use
- Our string library has also historically been deficient

## This Effort:

- Define a record-based *string* type
- Close string leaks
- Convert string literals to type *string* (were previously *c_string)*
- Support a modern set of library routines

# Strings: Status

## Status:

- Strings are significantly improved for version 1.13
  - now implemented as records
  - virtually leak-free, due to memory management fixes (see next section)
  - performance of strings has also improved
- Strings only support ASCII at present
  - want to support UTF-8 as well, and by default

# Strings: Record-based Definition

- **The string type currently looks like:**

```
record string {
  var len: int = 0;                    // length of the string, in bytes
  var size: int = 0;                   // size of buffer pointed to by buff
  var buff: c_ptr(uint(8))= nil;       // buffer to store the string
  var owned: bool = true;              // deallocate buff on scope exit?
  var locale_id = chpl_nodeID;         // locale where buff is allocated
}
```

- **Implemented in Chapel modules, used by the compiler**
  - Compiler hooks onto this type early in compilation
  - Alternative implementations could easily be swapped in
- **Permitted us to remove special cases in the compiler**
  - *string* is now handled like other records
    - modulo string-specific features like literals and param support
  - Removed coercions from *c_string* to *string*

# Strings: literals (`“text”`) now have type *string*

- **String literals were of type *c_string***
  - Reason: lack of support for param records in the compiler
    - supporting param *c_strings* seemed more straightforward than *string*
  - Required implicit coercions to *string* in many cases

    ```
    var x = “Hello, World”;    // implicit runtime coercion from c_string to string
    ```
  - *c_string* isn’t a primary Chapel type, so this was unattractive
- **Made string literals be of type *string***
  - *string* literals are now constructed at program startup
    - stored as locale-private global variables
  - Implicit coercions are no longer necessary
  - Introduced a new literal format for C strings (for interoperability only):

    ```
    var my_c_string = c“Hello, World”;
    ```
- **Added support for *param strings***
  - Specific to *string*; general *param* records remain future work

# Strings: Add new library routines

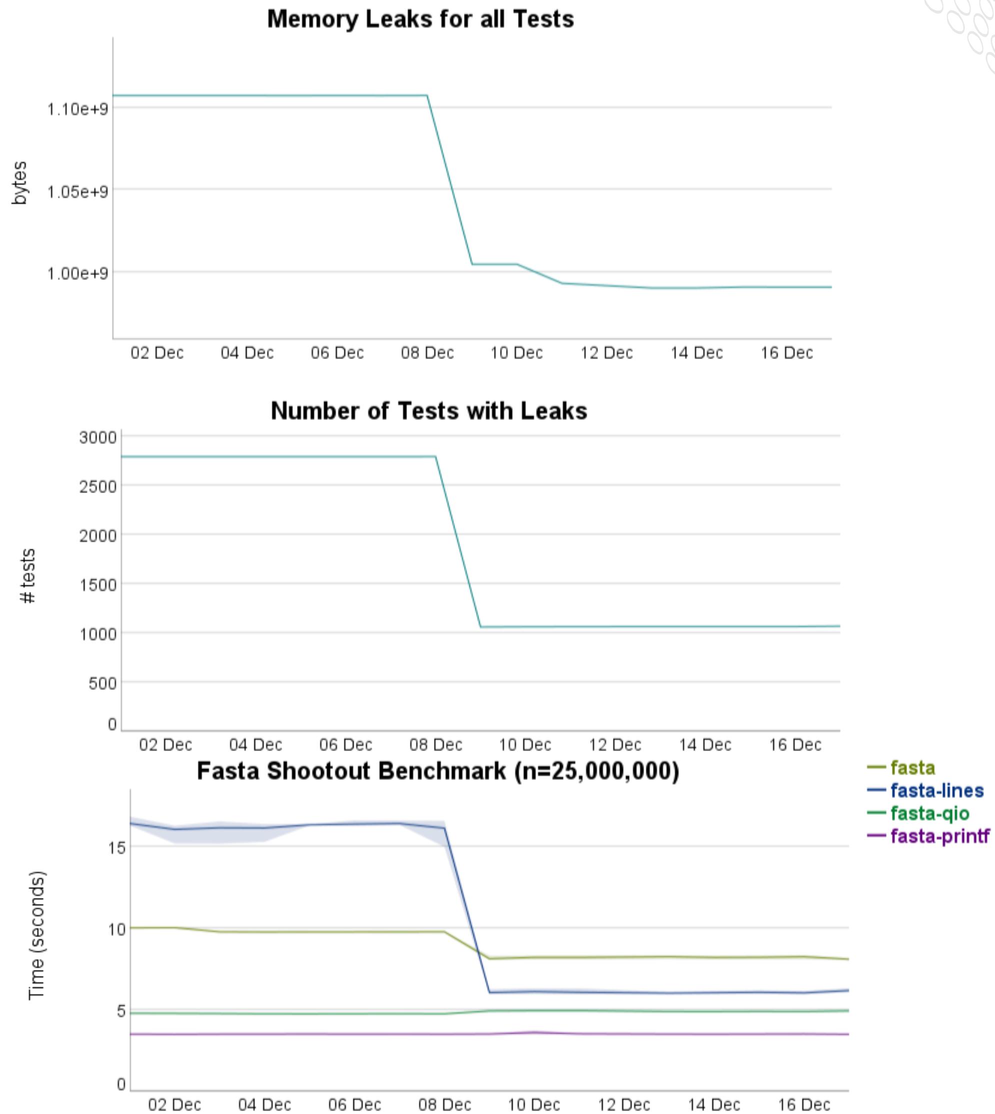
- **Current string library routines:**

```
this()        // substring          isEmptyString()
these()       // iterate over chars isUpper()
startsWith()                        isLower()
endsWith()                          isSpace()
find()                              isAlpha()
rfind()                             isDigit()
count()                             isAlnum()
replace()                           isPrintable()
split()                             isTitle()
join()                              toLower()
strip()                             toUpper()
partition()                         toTitle()
localize()                          capitalize()
c_str()                             +, +=, *, ==, !=, <=, ...
```

# Strings: Impact

**December 9th testing reflects the switch to string records:**

- **string-based memory leaks in our test suite went from 123MB to 22MB**
  - by 1.13, reduced to < 1 KB
- **# of tests with leaks in our suite was reduced by 2.6x**
- ***fasta-lines* version of fasta CLBG benchmark saw 2.7x speedup**
  - another version improved 20%



# Strings: Next steps

- **Add proper Unicode strings**
  - Current plan: Support multiple *string* encodings
    - focus on UTF-8 and ASCII only for now
  - Will require changes to string library interfaces/implementation
- **Performance and memory management improvements**
  - string representation
  - string library algorithms
  - general improvements for records
- **Continue to refine string library based on user feedback**

# Record Memory Management Improvements

# Record Memory Management: Background

- **Chapel's original design was overly naïve about records**
  - Original design was overly focused on plain-old-data (POD) records
  - Yet, in practice, records can be used to manage heap-based data
    - e.g., string buffers, array elements, class objects, C pointers, …
  - Such cases revealed inadequacies in our approach
    - no user-facing copy constructors
    - failure to distinguish initialization from assignment
    - inconsistent invocation of destructors
    - inappropriate conversion of references to intermediate values
  - Consequences:
    - memory leaks, use of stale references, double-frees
    - sequences of unnecessary copy/free pairs
  - Have been working to course-correct, spurred on by string work

# Record Memory Management: This Effort

- **Improve the implementation of records**
  - Eliminate errors when initializing, copying, destroying records
  - Ensure references to records are copied correctly
  - Lay foundation for extended constructor/destructor semantics

- **Chief catalyst: record-based strings**
  - Many other cases waiting in the wings as well:
    - arrays/domains (also special-cased today)
    - syncs/singles
    - barriers
    - GMP types
    - Random
    - ...
  - Building on records for these will
    - simplify the compiler
    - remove memory leaks
    - improve the user experience (e.g., no need to free barriers)

# Record Memory Management: Possible Issues

- **Core issue:**
  - Failure to handle construction/assignment/destruction correctly
- **Overly aggressive autoCopy/autoDestory optimization**
  - These operations can be costly
  - Compiler uses memcpy() where it ought to be using assignment ops
  - Errors lead to leaks or the use of stale buffer handles
- **Creation of false aliases**
  - Wide references incorrectly converted to intermediate local values
  - Identity of original string is lost
- **Unnecessary, but valid, sequences of copy/destroy pairs**
  - b = copy(a); destroy(a); c = copy(b); destroy(b); ....
  - Commonly occur at function exits
  - Particularly evident after inlining

# Record Memory Management: In Principle

- **Using strings as motivation:**

```
record string {
  var buff  : c_ptr(uint(8)) = nil;          // A class
  var owned : bool           = true;         // Enable shallow copy
  ...
}

proc foo(str0 : string) {     // str0 defaults to const ref
  var str1 = "Hi";            // str1.buff initialized to a copy of "Hi"


  str1 = bar(str0);           // assignment called; str1.buff reclaimed
                              // str1 mem-copied from return temp

  return str1;                // str1 returned to caller

}
```

# Record Memory Management: In Practice*

- **Using strings as motivation:**

```
record string {
  var buff  : c_ptr(uint(8)) = nil;          // A class
  var owned : bool           = true;         // Enable shallow copy
  ...
}

proc foo(str0 : string) {      // str0 defaults to const ref
  var str1 = "Hi";             // str1.buff initialized to a copy of "Hi"
  var tmp0 : string = str0;    // str0 is de-referenced to a compiler temp

  str1 = bar(tmp0);            // memcpy() called instead of assignment
                               // str1.buff leaked as a result
  var ret0          = str1;    // return temp inserted; ret0 a copy of str1
  return ret0;                 // ret0 returned to caller
                               // revised str1.buff is reclaimed
}
```

(* while such errors did occur in practice, they typically wouldn't for so simple an example)

# Record Memory Management: Approach

- **Compiler supports many record-like types**
  - Records, tuples, ranges, iterators, arrays, ...
  - Chapel's initial focus had been on parallelism and array performance
    - Tolerated leaks during early development — but no longer
  - Challenging to revise logic for strings without harming arrays
- **Developed stress tests for records**
- **Initially, developed specialized code paths for string**
  - Refactored logic for returning records by value
  - Tweaked logic for autoCopy/autoDestroy
  - Tweaked module code for String
  - Ensured new logic would extend to other record types
- **Later, extended to general user-defined records**

# Record Memory Management: String Impacts

- **Record-based strings integrated on Dec 8, 2015**
  - No correctness regressions for pre-existing tests
    - Minor errors in some stress tests eliminated over subsequent 2 weeks
  - Resulted in significant memory leak improvements:

| | Before | After | Delta | Impact |
|---|---|---|---|---|
| Tests with leaks | 2,798 | 1,064 | -1,734 | -62.0% |
| Bytes leaked (MiB) | 1,081 | 981 | -100 | -9.2% |
| String copy data (MiB) | 111 | 21 | -90 | -80.8% |
| String concat data (MiB) | 10 | 0 | -10 | -100.0% |

- **String copy data reduced to 300 bytes by late Feb 2016**
  - Result of several small changes

# Record Memory Management: Status

- **New code paths apply to all user-defined records**
  - Strings no longer special-cased in this regard
  - Several other implementation-oriented records are still special-cased:
    - ranges, domains, arrays, distributions, tuples, iterators, …

- **Certain deficiencies remain**
  - Conversion of wide references to values can still occur
  - Extraneous sequences autoCopy/autoDestroy pairs remain

# Record Memory Management: Next Steps

- **Make implementation leak-free**
- **Remove other special cases**
  - tuples, ranges, domains, arrays, …
- **Rewrap existing types to leverage record improvements**
  - sync/single, barrier, GMP, Random, …
- **Finalize revised specification of record semantics**
  - initializers (constructors) and destructors
  - record copies
  - returning records from functions
- **Bring implementation in-line with specification**
  - Implement user-facing features (e.g., initializers)
  - Eliminate unnecessary copies
  - Fix wide reference-to-record issue

# Use Statement Improvements ('`use`')

# Use Statement: Background

- **Use statement brought in all visible symbols from module**
  - No way to selectively control which symbols were imported
    - Led to shadowing of outer-scoped variables with same name
    - Led to conflicts if two ‘use’d modules defined symbols with same name
    - Could fully qualify names to disambiguate, but fairly heavy-weight

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M;
    foo();
    var a = bar;    // Finds M.bar, rather than myMod.bar
  }
}
```

```
module M {
 var bar = 13;
 proc foo() { … }
}
```

# Use Statement: Import Control

- **Add import control for use statements**
  - 'except' keyword prevents unqualified access to symbols in list

    ```
    use M except bar;  // All of M's symbols other than bar can be named directly
    ```
  - 'only' keyword limits unqualified access to symbols in list

    ```
    use M only foo;    // Only M's foo can be named directly
    ```
  - Permits user to avoid importing unnecessary symbols
    - Including symbols which cause conflicts

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo;
    foo();
    var a = bar;   // Now finds myMod.bar, rather than M.bar
  }
}
```

```
module M {
 var bar = 13;
 proc foo() { … }
}
```

# Use Statement: Symbol Renaming

- **Add ability to rename imported symbols**

```
use M only bar as barM;
```

  - Allows users to avoid…

    …naming conflicts between multiple used modules

    …shadowing outer variables with same name

  …while still making that symbol available for access

```
module M {
 var bar = 13;
 proc foo() { … }
}
```

```
module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo, bar as barM;
    foo();
    var a = bar;     // Still finds myMod.bar, rather than M.bar
    var b = barM;    // refers to M.bar
} }
```

# Use Statement: Application to enums

- **'use' is now applicable to enums**
  - Frequently requested feature from users
    - by default, unqualified access not supported to protect namespace
  - Supports direct references to an enum's symbols

```
enum color {red, blue, yellow};
use color;
var foo = blue; // instead of color.blue
```

# Use Statement: Status and Next Steps

## Status:

- All these features now available as of 1.13
- Cleaned up compiler representation of use statements

## Next Steps:

- Support a means of requiring all accesses to be qualified
  - e.g., `use M except *;` or `use M only ;`
- Allow module-private use statements
  - Today's uses are transitive
    - symbols used by another used module are transitively accessible
  - Anticipate challenges relative to point-of-instantiation for generics
- Warnings for uses that import…

  …globals with types hidden by 'only' or 'except' lists

  …functions that take arguments of hidden types

  …functions with return types that import list hid
- Public/private type definitions and class/record members

# Improving Reduce Intent Support

# Reduce Intent Support: Background

- Reduce intents were introduced in 1.11

```
var tmin, tmax, ttot: real;
forall i in 1..numTrials with (min reduce tmin,
                               max reduce tmax,
                                 + reduce ttot) {
  const start = getCurrentTime();
  …          // do some computation here
  const elapsed = getCurrentTime() - start;
  tmin = min(tmin, elapsed);
  tmax = max(tmax, elapsed);
  ttot += elapsed;
}
```

*Create task-local copies of tmin, tmax, ttot. Reduce at task-end using min, max, +.*

- They had some limitations:
  - only certain operators were supported:
    + * && || & | ^
  - could only be applied to forall-loops, not coforalls or cobegins

# Reduce Intent Support: This Effort and Impact

**This Effort:** Added ability to use reduce intents with…

…min, max ops; simple user-defined reductions

```
var r: real;
forall a in A with (max reduce r) do
  r = max(r,a);
```

*`max` reduction over `A` into `r`*

…coforall loops

```
var r: real;
coforall t in MyTasks with (+ reduce r) do
  r += t;
```

*plus reduction using **`coforall`***

**Impact:** More opportunities to use reduce intents now

- can be more natural/efficient than reduce operators for many cases
- e.g., ISx benchmark
  - gathers timing statistics using min, max, + reduce intents on a single loop
  - more elegant and efficient

# Reduce Intent Support: Next Steps

- Design and implement support for more complex cases:
  - expressing accumulation step in a general way

```
use ReductionLibrary;
forall a in A with (myReduceOp reduce r) do
  r = ???(r,a);
```

    - one candidate:

```
  r reduce= a;
```

*How to express accumulation for a black-box reduction?*
*Ideally, approach would be independent of reduction op*

  - element type differs from intermediate data and/or reduction result
    - e.g. min-k reduction
      - elements: real
      - intermediate data and result: k-tuple of reals
    - e.g. "is sorted" reduction
      - elements: real
      - result: bool "are all elements sorted?"
      - intermediate data: (something more complex than a real or bool)
- Reduce intents for cobegin statements

# Ref Return Intent Changes

# Ref Return: Background

## Background:

- ‘ref’ return intent originally introduced to support array accessors
  - used ‘var’ keyword at that point; ‘ref’ had not yet been introduced
- some accessors need to distinguish between reads and writes
  - e.g., sparse arrays can be read anywhere, but only written at “non-zeroes”
- supported this via a compiler-provided ‘setter’ param
  - cloned functions for setter vs. getter cases; chose clone based on callsite

```
proc mySparseArray.this(i: int) ref {
  if !explicitlyStored(i) {
    if setter then
      halt(“trying to write to a non-zero location: ”, i);
    else
      return 0;
  } else {
      return data[i];
  }
}
```

# Ref Return: More Background

- **‘setter’ always felt like a wart in the language**
  - clunky, invisible local param, provided only for ref-return functions
- **‘ref’ return intents resulted in unnecessary copies**
  - counterintuitive
    - “I said ‘return-by-ref’, why would a copy be made?”
  - so why were they added?
    - compiler cloned such functions to create getter vs. setter versions
    - ‘getter’ versions always returned by value, so added copies
  - user had no means to change this behavior
    - i.e., no means to specify a different return intent for the getter function
  - one result: excess copies/destroys for array accesses
    - e.g., in one case, 4 layers of ‘ref’ calls resulted in 4 such copy/destroy pairs
    - hurt performance for arrays of nontrivial types (e.g., strings, records)

# Ref Return: This Effort

- **Replaced 'setter' with a more principled approach:**
  - Support function overloads differentiated only by return intent
  - call 'ref' return intent version in l-value contexts
    - otherwise, call other version
    - at first, may seem odd to resolve based on callsite…
      …yet this is what we were effectively doing for 'setter' already
- **Added a new 'const ref' return intent**
  - Supports 'ref' vs. 'const ref' overloads for l-value vs. r-value contexts
  - Or simply creating functions with 'const ref' return intent…
- **For more information, see documentation:**
  - Language specification:
    See "Return Intents" (sections 13.7.1 – 13.7.3 in version 0.981 of spec)
  - Language evolution page:
    http://chapel.cray.com/docs/1.13/language/evolution.html#ref-return-intent

# Ref Return: Example

- **Motivating example, revised:**

```
proc mySparseArray.this(i: int) ref {            // "setter" equivalent
  if !explicitlyStored(i) {
    halt("trying to write to non-zero location");
  else
    return data[i];
}
proc mySparseArray.this(i: int) const ref {      // "getter" equivalent
  if !explicitlyStored(i) {
    return 0;
  else
    return data[i];
}

SpsArr[1] = 2.3;        // writes call ref (black) version
writeln(SpsArr[1]);     // reads call const ref (blue) version
```

# Ref Return: Impact and Next Steps

## Impact:

- removes the ‘setter’ wart from the language
- removes compiler clones of ‘ref’ return functions
- permits unnecessary copies to be avoided
- cases that used ‘setter’ must be rewritten as overloads
  - yet, common case of ‘ref’ return without ‘setter’ case requires no changes
- avoids extraneous copies for array accesses, “getter” cases

## Next Steps:

- Fix a bug in which the wrong overload is invoked for returned arrays
  - ‘ref’ version is called in an r-value context
    - harmless in many cases
    - yet, in motivating example, would lead to an incorrect halt
  - bug was only found recently, but almost certainly existed with ‘setter’ as well

# Passing Synchronization Variables to Generics

# Passing Syncs to Generics

## Background:

- Passing syncs to generics "unwrapped" them in certain cases:

```
proc foo(const x) { writeln(isSync(x)); }
var s$: sync int = 42;
foo(s$);                    // acted like foo(s$.readFE()) and printed 'false'
```

- Historically, believed that such unwrapping was natural & useful
  - over time, this has seemed increasingly surprising

## This Effort:

- Make generic sync arguments behave more like one would expect
  - e.g., example above now behaves equivalently to:

```
proc foo(const x: sync int) { writeln(isSync(x)); }
var s$: sync int = 42;
foo(s$);                    // doesn't unwrap 's$' and prints 'true'
```

## Impact / Status:

- Synchronization variables are now far more principled than before

# Simple Synchronization Types Only

# Simple Sync Types Only

## Background:

- Historically, Chapel permitted any type to be declared ‘sync’/’single’
  - This was thought to be attractively general and orthogonal — for example:

```
var A$: [1..n] sync int;     // an array of synchronized integers
var B$: sync [1..n] int;     // a synchronized array of integers
```

  - Synchronized arrays could be interpreted sensibly in some cases:

```
B$ = 0;                      // block until B$ is empty; zero; leave full
```

  - But others are less clear:

```
B$[3] = 1;                   // how should the full/empty bit be involved?
```

  - Records, complexes have similar issues
- Some time ago, decided sync/single should support simple types only
  - effectively, ones with a single logical value
  - atomic variables already follow a similar philosophy
- Documented decision in the spec, but never enforced it in the compiler

# Simple Sync Types Only

## This Effort:

- Added enforcement to the compiler to restrict sync/single to:
  - bools
  - ints / uints
  - reals / imags
  - strings
  - classes
    - since class variables are object references, they're similarly "simple"

## Impact/Status:

- Removes a dark semantic corner from the language / implementation
- Makes syncs/singles more like atomics

## Next Steps:

- Consider adding support for atomic class variables

# Simple Sync Types Only: Sidebar on *sync void*

## Motivation:

- Some cases want full/empty semantics without an associated value

```
var flag$: sync bool;     // we often declare variables like this where we don't
                          // care about the bool, just the full-empty state
```

- 'sync void' may be an appropriate type for such cases

## Status:

- the 'void' type doesn't work well in general
  - have long-term plans to use it to optimize away variables / fields

```
var stride: if stridable then int else void;
```

- this effort does not explicitly prohibit 'sync void'
  - yet, like 'void', it doesn't work well enough to be useful either

## Next Steps:

- Complete implementation of 'void' type
- Make 'sync void' usable

# Casting Types to Strings

# Casting Types to Strings

## Background:

- It's often useful to convert a type to a string
- In the past, has been done using a helper function, typeToString():

```
writeln("x's type is: ", typeToString(x.type));
```

## This Effort:

- Belated insight: Why not support via casts from types to strings?

```
writeln("x's type is: ", (x.type):string);   // the parens are optional
```

## Impact/Status:

- Unification of design; permits us to retire a lame helper function
- typeToString() still supported in 1.13 but gives a deprecation warning

## Next Steps:

- Open question about whether varargs should support type/val mixes
  - This is why writeln() can't simply accept types as arbitrary arguments today:

```
writeln("x's type is: ", x.type);
```

# Improved min/max Type Signatures

# Improved min/max Type Signatures

## Background:

- Traditionally, min and max have been defined completely generically:

```
proc min(x, y) { return if (x < y) then x else y; }
```

- This causes problems for certain type signatures:

```
min(myInt32, myUint32);            // error: can't resolve min's return type
```

- Other operators support formal type arguments to avoid this:

```
proc +(x: int(?w), y: int(w)) { … }      // and similarly for other types
```

## This Effort:

- Declare 'min'/'max' signatures like other operators (and keep generics)

## Status/Impact:

- 'min'/'max' now behave more uniformly with respect to other operators

## Next Steps:

- Make 'min' / 'max' even more like other operators?

```
smallest min= newValue;
```

# Comparison Operators on int/uint Pairs

# Comparison Ops on int/uint Pairs

## Background:

- Chapel does not support operators on int / uint pairs by default

```
proc +(x: int, y: uint) { … // throws a compile-time error }
```

  - rationale: It's ambiguous whether the result should be an int or a uint
    - It's not hard to construct scenarios that would most naturally prefer either result
    - So, handle such cases via casting, user overloads, library-based overloads, or …
- This rule was naively applied to comparison operators as well:

```
proc <(x: int, y: uint) { … // throws a compile-time error }
```

## This Effort:

- Realized that comparisons don't have this problem, so added them
  - results are boolean
  - comparison can be computed despite differing value ranges

```
proc <(x: int, y: uint) {
  if x < 0 then return true;
  return (x:uint) < y;    // cast is safe due to previous conditional
}   // in practice, this is implemented more concisely via short-circuiting operators
```

# Comparison Ops on int/uint Pairs

## Status / Impact:

- Integer comparisons are now much simpler to express without casts

## Next Steps:

- Provide standard libraries of int/uint overloads for other operators
  - Gives users the ability to select the semantics they want for a given scope
    - “int wins”
    - “uint wins”
    - “first argument wins”
    - …others?

# locale.numPUs()

# locale.numPUs() replaces numCores

**Background:** locale.numCores was inaccurate and insufficient

- actually returned number of hardware threads, not number of cores
- value always reflected OS limiting; physical count not available

**This Effort:** Provide a better interface

```
proc numPUs(logical: bool=false, accessible: bool=true)
```

- resolves both issues: can return…
  …cores or hardware threads (logical=false vs. true)
  …accessible or physical (accessible=true vs. false)
- deprecate numCores

**Impact:** Provides a richer and more accurate interface

**Next Steps:** Improve accuracy of returned values

- certain OS accessibility patterns cause incorrect counts

# Other Language Improvements

# I/O Improvements

- **deprecated Readers and Writers in favor of channels**
- **default I/O routines now ignore ‘param’ fields, like ‘type’**
- **channel.readbits/writebits accept additional integer types**
- **removed Chapel I/O support for *c_strings***
  - rationale:
    - led to problems across multi-locale settings
    - *c_string* intended for interoperability, not general Chapel operations

# Other Language Improvements

- **added support for subclassing generic classes**
- **added support for mixed int/uint range slicing**
- **added casting of fully-qualified enum strings to values**
  (contributed by Nick Park)
  - e.g., the following now works: `..."color.red":color...`
- **added support for comparing c_void_ptr to nil**
  (contributed by Nick Park)
- **added support for implicitly coercing c_ptr to c_void_ptr**
- **added scalar versions of domain.exterior(), interior(), …**

# Legal Disclaimer

*Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.*

*Cray Inc. may make changes to specifications and product descriptions at any time, without notice.*

*All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.*

*Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.*

*Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.*

*Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.*

*The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.*

CRAY
CHAPEL

CRAY
THE SUPERCOMPUTER COMPANY