Language Improvements

Chapel version 1.19

March 21, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- Initializers
- Multibyte Strings and Unicode
- Error Handling
- Delete-free Programming
- Ongoing Effort: Nilability
- 'param' Floating-Point Values
- 'forall' vs. '[]' loops
- Shape / Index Preservation
- Numeric Literals with Underscores
- String to Numeric Casts
- Record =, ==, !=
- New Reserved Words

Initializers

- [The 'init=' method]
- [New-Expressions with Type Aliases]
- [Invocation of Default Initialization]
- [Improvements to Error Messages]

The 'init=' Method

The 'init=' Method: Background

- Wanted users to be able to define initialization of a variable from a value:

```
var x: MyBigInt = 5;    // MyBigInt is user-defined
```

- This pattern was unintuitive or impossible for users to implement
  - compiler generated default-initialization + assignment
  - some types won't support assignment

```
var x: MyBigInt = 5;
// used to become...
x.init();
x = 5;
```

The 'init=' Method: Background

- This pattern prevented initialization of atomics from int/real/bool values

```
var x: atomic int = 5;
// used to become...
x.init();
x = 5;  // compile-time error! atomics do not support assignment
```

The 'init=' Method: Background

- A simple idea: Invoke 'init' with given expression

  problem: may enable unintended/confusing initialization patterns

```
proc IntList.init(length: int) { }
var x: IntList = new IntList(5);  // x.init(5): create a list of length 5
var x: IntList = 5;  // also x.init(5), but was not intended to be legal
```

The 'init=' Method: Background

- In some cases, compiler assumed single-arg initializers were copy-initializers
  - as a result, a 'where' clause was required on certain initializers

```
record ClassWrapper { type T; var cls: T; }

// Enable 'new ClassWrapper(T)', prevent usage as copy-initializer
proc ClassWrapper.init(cls: ?T)
  where !isSubtype(T, ClassWrapper)
{ ... }
```

The 'init=' Method: This Effort

- Introduced a new initializer form named 'init='
  - enables variable initialization from arbitrary expressions
  - replaces 'init' as the copy-initializer

- The 'init=' method is opt-in for the Chapel 1.19 release
  - compiler will still use 'init' for copy initialization if there is no 'init='
  - no compile-time warnings for use of 'init' for copy initialization

- Updated internal/standard modules to use 'init=' when possible

The 'init=' Method: General Details

- 'init=' has constraints similar to 'init'
  - e.g. fields must be initialized in declaration order
- 'init=' may only have one argument
- 'init=' can invoke other initializers via 'this.init(...)'
  - in 1.19 cannot invoke other 'init=' methods
- 'init=' is only invoked by the compiler, and only in two cases
  - copy initialization, e.g.: `var myRec = otherRec;`
  - initializating a variable from an expression: `var x: atomic int = 5;`

The 'init=' Method: Non-Generic Types

- Compiler-generated 'init=' accepts one argument of the same type
  - for non-generic types, the method signature is simple:

```
record R {
  var x: int;
}

proc R.init=(other: R) {
  this.x = other.x;          // initialize each field of 'this' from 'other'
}
```

The 'init=' Method: Non-Generic Types

- Users can provide their own 'init=' and keep the compiler-generated 'init='
  - unlike 'init' where a user-defined 'init' disables the compiler-generated 'init'

```
record R { var x: int; }

proc R.init=(val: int) {
  this.x = val;
}

var A = new R(5);   // A.init(5)      compiler-generated 'init'
var B = 10;         // B.init=(10)    user-defined init=
var C = A;          // C.init=(A)     compiler-generated init=
```

The 'init=' Method: Generic Types

Problem: 'init=' for generic types requires knowing the intended instantiation

```
record R {
  type T;
  var x: T;
}
var x: R(real);
var y: R(int) = x; // should be an error
```

'R' as an argument type is not enough.
Where would 'int' come from?

```
proc R.init=(other: R) { ... }
```

The 'init=' Method: Generic Types

Solution: Allow querying 'this.type' in 'init=' methods:

```
// compiler-generated 'init=' for 'R'
proc R.init=(other: this.type) {
  this.T = other.T;  // serves as assertion, may be unnecessary in future
  this.x = other.x;
}

var x: R(real);
var y: R(int) = x;  // error! cannot copy-initialize R(int) from R(real)
```

The 'init=' Method: Using 'this.type'

- In some cases inferring types from the 'init=' argument may be insufficient

```
record Wrapper { type T; var x: T; }

// Goal: Initialize 'Wrapper(T)' from 'T'
// A simple first attempt: infer 'T' from the argument
proc Wrapper.init=(value: ?T) { ... }

// The intended instantiation may be different from the given value
// Note: '5' is an 'int(64)' by default!
var x: Wrapper(int(8)) = 5;  // error! tries to instantiate Wrapper(int(64))
```

The 'init=' Method: Using 'this.type'

- The 'this.type' query can be used to constrain the 'init=' argument

```
// Solution: constrain the argument with 'this.type.T'
proc Wrapper.init=(value: this.type.T) { ... }

// Now coerces '5' to 'int(8)' as with normal methods
var x: Wrapper(int(8)) = 5;
```

- In practice, necessary to enable atomic variable initialization:

```
var x: atomic int(8) = 5;
```

The 'init=' Method: Impact

- Leveraged 'init=' to enable initialization of atomics from values
  - also used 'init=' for bigint, instead of assignment operator

- Enables powerful, principled initialization patterns for users
  - no longer need to rely on assignment operator for initialization

- An initializer can no longer be mistaken for a copy-initializer

The 'init=' Method: Status

- 'init=' is used within internal/standard/package modules and in test suite

- Optional feature in Chapel 1.19
  - 'init' methods still work for copy-initialization

The 'init=' Method: Next Steps

- Finalize how 'this.type' can be used
  - can users write 'this.T' instead of 'this.type.T' ?
  - how will 'this.type' interact with partial instantiations?

- Leverage 'init=' for arrays

- Enable 'init=' by default

- Explore support for 'this.init=(...)' inside an 'init='

New-Expressions and Generic Type Aliases

New-Exprs and Aliases: Background

- Instantiated type aliases could be useful in new-expressions
  - minimizes keystrokes for instantiating the same type many times
  - easier to find/change a frequently-used type

```
record R {
  type T;
  var x: T;
  type U;
  var y: U;
}
type RIR = R(int, real);
var x = new RIR(5, 10.0);  // not allowed in 1.18
```

New-Exprs and Aliases: This Effort

- Enabled usage of type aliases in new-expressions via named-expressions
  - for each generic field, an implicit named-expression in the new-expression

```
type RIR = R(int, real);
var x = new RIR(5, 10.0); // new R(T=int, U=real, 5, 10.0)
```

- Not currently supported for types with fully-generic fields (e.g. 'var x;')
  - still exploring options for supporting in a principled manner

New-Exprs and Aliases: Why Named Exprs?

- Tradeoff between named-expressions and positional arguments
  - positional arguments would require fields to be in a certain order
  - named-expressions require initializer arguments to have specific names

- Named-expressions are considered to be more flexible
  - fields and initializer arguments can be in any order
  - common for initializer arguments to have the same name as fields
  - can take advantage of existing compiler-generated initializer signature

New-Exprs and Aliases: Status, Next Steps

Status:

- Some type aliases can be used in new-expressions in 1.19
- Not supported for types with fully-generic fields (e.g. 'var x;')

Next Steps:

- Support for aliases with fully-generic fields
  - explore feasibility of 'this.type' queries in such cases

Invocation of Default Initialization

Default Initialization: Background

- "Default initialization" occurs when a variable is declared without an expression
  - concrete types result in a call to 'init' without arguments

```
var x: R; // x.init();
```

- Generic types require passing instantiation information to initializer
  - in 1.18 there was a difference between user & compiler-generated initializers

```
var x: R(int, real);
// user initializer: x.init(int, real);
// compiler-generated: x.init(T=int, U=real);
```

Default Initialization: This Effort

- Always invoke default-initialization with named-expressions
  - eliminates inconsistency between user/compiler-generated initializers
  - named-expressions are considered more flexible than positional arguments

```
var x : R(int, real);
// now, in every case: x.init(T=int, U=real);
```

Default Initialization: Status, Next Steps

Status:

- Present in 1.19 release
- Minimal impact expected: changing initializer argument names

Next Steps:

- Unify approach with type-aliases used in new-expressions
  - default-initialization supports fully-generic fields,
    type-aliases in new-expressions do not (yet)

Improved Initializer Error Messages

Initializer Error Messages

- **Background:**
  - In 1.18 compiler issued warning for user-defined constructors
  - In 1.18 new-expressions without argument list could result in 'nil'

```
var x = new owned C; // x == nil
```

- **This Effort:**
  - In 1.19 compiler issues error for user-defined constructors

```
Constructors have been deprecated as of Chapel 1.18. Please use
initializers instead
```

  - In 1.19 compiler issues error for new-expressions without argument list

```
type in 'new' expression is missing its argument list
```

Initializer Improvements: Summary

- Variable initialization is significantly more powerful through 'init='
- Instantiated type aliases can now be used in new-expressions
- Default-initialization is better defined than in 1.18
- Error messages continue to improve
- Initializer design is nearly finalized

Multibyte Strings and Unicode

Strings: Background

- Chapel supports UTF-8 Unicode strings
  - currently requires POSIX locale environment variables set to a UTF-8 locale
- Chapel 1.18 supported:
  - UTF-8 string literals

```
var str = "événement";
```

  - I/O of UTF-8 characters
  - string indexing by byte or by codepoint

```
str[i: codepointIndex]; // returns i'th codepoint
```

  - a variety of methods on UTF-8 strings, e.g. 'isAlpha()', 'split()', 'find()', ...

Strings: UTF-8

- UTF-8 is a common multibyte character set
  - one to four bytes per character
- Every valid ASCII character is a valid UTF-8 character
  - with the same meaning
- A complete multibyte UTF-8 character describes a Unicode *codepoint*
  - Unicode is currently a 21-bit character set
- It is possible to combine certain codepoints in the same printing position
  - the result is a *grapheme*
  - example: e + ´ = é (though in this case a single codepoint for é exists)

Strings: Indexing

- Indexing by byte is fast
  - fixed width, random access

- Indexing by codepoint is slow
  - variable width, count each multibyte character forward from beginning

- Indexing by graphemes would add an extra layer of variable width

Strings: This Effort

- Added support for slicing a string by a range of codepoint indices

```
str[3:codepointIndex..]
```

- Adjusted string indexing to always return a string
  - previously it returned an integer if codepoint indexing was used

- Documented the environment variables necessary to enable UTF-8 support

Strings: This Effort

- Added 'byte', 'bytes', 'codepoint', and 'codepoints' methods that return integers

```
var str = "événement"; // In UTF-8, c3 a9 76 c3 a9 6e 65 6d 65 6e 74
var chr: uint(8) = str.byte(3); // results in 0x76 aka 'v'

for c in str.bytes() do
  // manipulate each byte as a uint(8)

var cpt: int(32) = str.codepoint(3); // results in 0xe9 aka 'é'

for cp in str.codepoints() do
  // manipulate each codepoint as an int(32)
```

Strings: This Effort

- Made several high-level design decisions about strings
  - support only the UTF-8 character encoding in ordinary strings
  - create another string-like type that can also hold binary data (e.g., ’bytes’)
  - allow indexing explicitly by byte or codepoint
  - default indexing will be by codepoint
  - continue to support ctype character classes ('isAlpha()', 'isUpper()', etc.)
    - no detailed Unicode character properties, at least for now
  - no indexing and iteration by grapheme, for now
    - but avoid precluding this in the future
  - deprecate 'ascii()' and 'asciiToString()' in the next release

Strings: Impact

- More string functionality is available
- The string API is more regular
- Future direction is known
- Idiom 'ascii(str[i])' now has a faster replacement

```
str.byte(i)    // avoids creating a string temporary
```

Strings: Next Steps

- Augment the string implementation to incorporate the new design decisions:
  - use UTF-8 encoding in strings no matter what the POSIX locale is
  - create a string-like type that can hold arbitrary binary data
  - add 'byteIndex' as an alternative to 'codepointIndex'
  - adjust indexing, iteration, and slicing to assume codepoint indices
  - deprecate 'ascii()' and 'asciiToString()'
- Document that Chapel source code is UTF-8

Strings: More Design Questions

- How and when are errors with invalid UTF-8 sequences reported?
- How to handle POSIX filenames?
  - filenames are not necessarily UTF-8 but may often be
- Should the I/O system support conversion between character sets?
  - would address garbles when printing UTF-8 data to a non-UTF-8 terminal
- Should Chapel source code allow non-ascii identifiers? e.g.

```
var événement = 1;
```

Error Handling

Error Handling: Background

- Error handling has been recommended for use since 1.17
- Standard modules have been using error handling since 1.18
- Nonetheless, there has been a need for continuing work in this area:

  … to integrate the delete-free language design into error handling

  … to address bugs in the implementation

  … to improve documentation of throwing functions

Error Handling: Owned Errors – Background

- Error handling was added before 'owned' and 'shared'
- But error handling relies heavily on subclasses of Error
- Resulted in several deficiencies:
  - 'throw new C' created an ‘unmanaged C’ but syntax implies ‘borrowed C’
  - double-delete when storing a caught error in a variable
  - double-delete when wrapping a caught error in another error

Error Handling: Owned Errors – Background

- Before this release, a try/catch block might look like this:

```
proc f() throws {
  throw new InvalidArgumentError();
}

try {
  f();
} catch e: InvalidArgumentError {
  throw new WrappedError(e); // led to double-free
}
```

Error Handling: Owned Errors – Background

- Before this release, a try/catch block might look like this:

```
proc f() throws {
  throw new InvalidArgumentError();
}

try {
  f();
} catch e: InvalidArgumentError {
  throw new WrappedError(e); // led to double-free
}
```

undecorated new is 'new borrowed' and can't be returned?

if 'e' is a borrowed Error, how can I transfer ownership?

Error Handling: Owned Errors – This Effort

- 'catch' now catches owned Errors:

```
try { ... }
catch e: MyError {
  // e has type 'owned MyError'
  globalError = e;     // transfers ownership to 'globalError', avoids double-free
}
```

- 'throw' now requires owned Errors:

```
throw new borrowed MyError();     // error: please throw 'owned'
throw new MyError();              // warning: please throw 'owned'
```

Error Handling: Owned Errors – This Effort

- Error handling now uses 'owned'

```
proc f() throws {
  throw new owned InvalidArgumentError();
}

try {
  f();
} catch e: InvalidArgumentError {
  throw new WrappedError(e);
}
```

now clear that ownership of the error is transferred out of f()

error ownership can now be transferred to WrappedError

Error Handling: Bug Fixes

- Addressed several error handling bugs this release
  - resolved memory errors when a function returning an array throws
  - addressed internal error for certain 'try!' patterns
  - fixed a problem with control flow analysis in functions with 'catch' blocks

Error Handling: Documentation – Background

- In 1.18, the generated documentation included 'throws' in the signature
  - details – what and when could be thrown – were integrated in the description

proc **copy**(*src: string, dest: string, metadata: bool = false*) throws

Copies the contents and permissions of the file indicated by *src* into the file or directory *dest*. If *dest* is a directory, will throw a FileNotFoundError. If *metadata* is set to *true*, will also copy the metadata (uid, gid, time of last access and time of modification) of the file to be copied. A partially copied file or directory may be present in *dest* if there is an error in copying.

Arguments:

- **src** : *string* -- The source file whose contents and permissions are to be copied
- **dest** : *string* -- The name of the destination file for the contents and permissions. May or may not exist previously, but will be overwritten if it did exist
- **metadata** : *bool* -- This argument indicates whether to copy metadata associated with the source file. It is set to *false* by default.

Error Handling: Documentation – This Effort

- Added ':throws <error>:' tag to chpldoc
  - similar to ':arg <name>:' tag, separates thrown errors from rest of description

```
/*
   :throws IsADirectoryError: when `dest` is directory.
   :throws SystemError: thrown to describe another error if it occurs.
*/
proc copy(src: string, dest: string, metadata: bool = false) throws {
```

proc **copy**(*src: string, dest: string, metadata: bool = false*) throws

Copies the contents and permissions of the file indicated by *src* into the file or directory *dest*. If *metadata* is set to *true*, will also copy the metadata (uid, gid, time of last access and time of modification) of the file to be copied. A partially copied file or directory may be present in *dest* if

Arguments:
- **src** : *string* -- The source file whose contents and permissions are to be copied
- **dest** : *string* -- The name of the destination file for the contents and permissions. May or may not exist previously, but will be overwritten if it did exist
- **metadata** : *bool* -- This argument indicates whether to copy metadata associated with the source file. It is set to *false* by default.

Throws:
- **IsADirectoryError** -- when *dest* is directory.
- **SystemError** -- thrown to describe another error if it occurs.

Error Handling: Impact, Next Steps

Impact: Error handling is significantly more robust

- Error handling now works harmoniously with delete-free
- Additional error handling patterns are enabled
- Language is now more stable in this area

Next Steps:

- Decide if Error should include a string field
  - and if 'new Error("error message")' should work
- Continue to improve documentation w.r.t. throwing routines
- Close memory leaks related to error handling

Delete-free Programming

Delete-free: Background

- Chapel 1.18 included language changes to enable delete-free programming
  - to avoid the need to remember to call 'delete'
  - to avoid certain memory errors
- Added 4 variants of class types:
  - 'owned', 'shared', 'borrowed' and 'unmanaged'
- Added compile-time lifetime checking
  - lifetime checker runs at compile-time
  - discovers certain memory errors
  - intentionally does not detect all memory errors

Delete-free: This Effort

- Fixed on-clauses over 'owned' and 'shared' class instances
- Fix bugs in the lifetime checker
- Added lifetime annotations
- Added compile-time checking for nil dereferences

Delete-free: Owned/Shared On-clause Fix

- Treat 'owned' and 'shared' similarly to 'borrowed' for locality

```
var instance: owned MyClass;  // instance pointer stored on locale 0
on Locales[1] {
  instance = new owned MyClass();  // allocate instance on locale 1
}
on instance {
  // which locale does this run on?
  // 1.18: locale 0
  // 1.19: locale 1
}
```

Delete-free: Lifetime Checker Bugs Fixed

- Fixed problems with lifetime checking within task constructs
- Improved lifetime checking within initializers
- Enabled lifetime checking for code at module scope
- Lifetime checking now handles iterators and loop expressions

Delete-free: Lifetime Clause

- Lifetime checker's default rules sometimes are not appropriate
- 'lifetime' keyword is now available to annotate a function
  - to override the defaults
  - to constrain lifetimes of arguments
- 'lifetime' keyword introduces a clause in some ways like a 'where' clause
  - with comma-separated parts

```
proc f(ref a, b, c) lifetime a=b, return c {
  a = b;
  return c;
}
```

'lifetime' clause

Delete-free: Returned Lifetime

```
class C { ... }
var global: borrowed C = ...;

proc getGlobalDefault(arg: borrowed C)
  return global;

proc getGlobal(arg: borrowed C)
  lifetime return global
  return global;
```

by default, the returned value has the lifetime of 'arg'

the lifetime clause indicates that the returned value has the lifetime of 'global'

Delete-free: Lifetime Constraints

```
record Collection {
  type elementType;
  var element: elementType;
}

proc Collection.addElementDefault(arg: elementType)
{ this.element = arg; }

proc Collection.addElement(arg: elementType)
 lifetime this < arg
{ this.element = arg; }
```

illegal by default:
the lifetime of 'arg' could be shorter
than 'this' and, by extension, 'this.element'

the lifetime clause requires 'arg'
to have a longer lifetime than 'this'

Delete-free: Compile-time Nil Checking

- Focuses on common errors, like lifetime checking

```
class MyClass {    proc method() { ... }    }

var obj: MyClass;          // obj is initialized to nil by default
obj.method();              // compile-time error: attempt to dereference nil

var x = new owned MyClass();
var y = x.release();       // now x stores nil
x.method();                // compile-time error: attempt to dereference nil
```

- Not intended to catch all errors at compile-time
  - to make it user-friendly in common cases

Delete-free: Impact, Next Steps

Impact:

- Delete-free language design is more stable
- Compile-time checking is more capable

Next Steps:

- Resolve open questions about delete-free language design
- Add nilable and not-nil class types

Delete-free Open Questions

Delete-free: Open Questions

- Should totally untyped arguments continue to instantiate as borrows?
  - should 'in' intent change the behavior here?
- Should we change the behavior of 'new C()' ?
- Should we change 'new borrowed C()' ?
- What should the receiver type be in a type method called from 'owned C' ?

Delete-free: Current Rules for Untyped Arguments

- A default-intent untyped formal instantiates to a borrowed type for any class-typed actual

```
f(new owned C());
proc f(x) {}              // x is a borrow
```

- Declaring a type overrides this behavior

```
g(new owned C());
proc g(y: owned) {}       // y takes over ownership from the actual arg
```

Delete-free: Overriding the Current Rules

- Experience is that sometimes this behavior needs to be overridden
  - even when a function does not know whether the argument will be owned
    - i.e. 'formal: owned' does not work in some cases
    - e.g. with a collection where the caller chooses between owned and shared

- Using an 'in'-intent enables ownership transfer without requiring a type

```
proc h(in z) {}
h(new owned C());      // z takes over ownership from the actual arg
h(new shared C());     // z shares the ownership with the actual arg
h(new borrowed C());   // z borrows the actual arg
```

Delete-free: Any Changes to the Current Rules?

- Should we keep the rule for untyped arguments?
- Should we keep the 'in'-intent exception?
- Should there be a different type-independent way to override it other than 'in' ? E.g.:

```
proc h(z: managed?) {}
h(new owned C());      // z takes over ownership from the actual arg
h(new shared C());     // z shares the ownership with the actual arg
h(new borrowed C());   // z borrows the actual arg
```

Delete-free: new C()

- Currently the same as 'new borrowed C()'
- Should we change it to 'new owned C()' ?
  - pro: result of 'new C()' could be returned or thrown
  - con: introduces asymmetry in type inference
- Even if 'new C()' generally means 'new borrowed C()', should we change it to mean 'new owned C()' in certain cases?
  - 'throw new C()'
  - 'this.field = new C()' in an initializer
  - 'myArray = [i in 1..n] new C()'

C is a class

Delete-free: new borrowed C()

- Currently 'new borrowed C()' is the same as (new owned C()).borrow()
- Should we keep this rule?
  - pro: symmetrical to other cases
  - pro: can be explained
  - con: may be unintuitive
  - con: the term 'borrowed' has a different meaning than in 'var x: borrowed C;'
- Should we keep it but discourage its use?
- Should we replace it with a different keyword?
  - e.g. 'new scoped C()'

C is a class

Delete-free: Type Methods on Classes

- Type methods on class C currently only work on 'borrowed C'
- We would like them to work with 'owned C' etc.
  - should the type of 'this' be 'owned' or 'borrowed'?

```
class C {
  proc type typemethod() {
    writeln(this:string);
  }
}
var x = new owned C();
x.type.typemethod();      // should it output 'owned C' or 'borrowed C' ?
```

Ongoing Effort: Nilability

Nilability: Background

- 'nil' pointers are problematic
  - Tony Hoare calls them "my billion-dollar mistake"
  - 'nil' dereference errors can be difficult to debug
  - programmers who practice defensive coding need to add nil checks
    - to ensure function behaves appropriately when passed any value
    - because compiler does not add runtime nil checks with '--fast'
- In Chapel, class instance pointers can currently be 'nil' and default to 'nil'

```
var x: MyClass;           // stores 'nil'
var y: owned MyClass;     // stores 'nil'
```

- Cf. 'ref' and 'const ref' variables always refer to a variable

Nilability: Other Languages

- Many current languages avoid 'nil' pointers
  - Swift
  - Rust
  - Scala
  - Kotlin
  - C# 8.0
- In these languages:
  - by default pointers cannot not store 'nil'
  - there is a way to opt-in to a nilable pointer (or an Option type)
- Should Chapel follow this trend?

Nilability: Why Types

- We considered whether nilability should be
  a) argument/return intent, or
  b) part of class types?
- Currently favor (b)
- This strategy enables important use cases:
  - creating an array of nilable or non-nilable classes
  - a generic data structure where caller indicates whether elements are nilable
  - generic identity function
- (b) is more similar to the approach used in other languages

Nilability: Nilable Class Types in Chapel

Proposal:

- A class type 'C' means a non-nil pointer to an instance
  - including 'borrowed C', 'owned C', 'shared C', 'unmanaged C'
- The type 'C?' is available to opt into being possibly 'nil'
  - including 'borrowed C?', 'owned C?', 'shared C?', 'unmanaged C?'
- 'C' and 'C?' are different types
- The ! operator unwraps a nilable value, halting if it is 'nil'

Nilability: Examples

```
proc getValue(x: C) {      // C is a class
  return x.value;          // no check needed here since 'x: C' cannot store nil
}

getValue(nil);             // compile-time error: 'getValue' expects a non-nilable
var a: C;                  // compile-time error: 'a: C' has no default value
var x: C?;                 // ok, use nil as the default value
getValue(x);               // compile-time error: x is nilable, passed to non-nilable
getValue(x!);              // compiles OK; adds a nil check at runtime
```

Nilability: Extensions

- We expect to add convenience features inspired by Swift

```
if let notNil = possiblyNil {
  // notNil has the non-nilable class type and cannot store nil
}

// if possiblyNil is nil, returns nil, otherwise computes someMethod()
possiblyNil?.someMethod()

// supplies a default value to use when possiblyNil is nil
possiblyNil ?? default
```

Nilability: Conditional Guard

- Should conditional guards introduce a new variable?
  - pro: each variable has a single type

```
if let notNil = possiblyNil {
  // notNil has the non-nilable class type and cannot store nil
}
```

- Or should the compiler just know that the variable is not nil inside the condition?
  - pro: uses existing syntax

```
if possiblyNil {
  // compiler knows that possiblyNil is not nil
}
```

Nilability: Next Steps

- Agree on initial language design direction and syntax
- Implement '?' and '!'
- Remove runtime checks for 'C'; leave them in for 'C!' even with --fast
- Add support for features inspired by the Swift conveniences
  - optional chaining e.g. 'possiblyNil.?someMethod()'
  - default operator e.g. 'possiblyNil ?? default'
  - conditional guard e.g. 'if let notNil = possiblyNil'

'param' Floating-Point Values

param real: Background

- 'param' expressions are computed at compile-time
- Enable succinct generic code and optimization
- Compiler views numeric literals as 'param'
- Chapel has supported param operations on integral types…

```
param y = 4 / 2;            // ok
```

... but not on 'real', 'imag', or 'complex':

```
param x = 1.0 / 2.0;    // compilation error
param y = 3.0 + 4.0i;   // compilation error
```

- Compiler intentionally shied away from such support
  - Primarily due to fear of distinct compile- vs. execution-time semantics
  - In part due to being non-expert in floating point

param real: This Effort

- Identified and addressed concerns about confusion
- What if compile-time result differs from run-time result?
  - IEEE 754 enables consistent results across languages and CPUs
    - including compile-time vs run-time
- Will users be confused by compile-time evaluation?
  - expressions like '1.0/2.0' already compile-time in C/C++
  - numerical analysts are likely to be accustomed to compile-time evaluation

param real: This Effort

- What about a customized rounding mode?
  - potential for confusion is limited by several factors:
    - 'param' expressions are relatively easy to identify
      - opt-in to them with 'param' arguments and return types
      - or expressions involving only literals
    - a hex float is reasonable for pre-computing a value a specific way
    - easy enough to write code to avoid 'param' evaluation

param real: This Effort

- Implemented support for floating-point 'param' operations
  - for 'complex', 'real', and 'imag' of all supported sizes

```
param a = 1.0: real(32);    // casting is now compile-time, and so are:
param b = -a;               //   unary + and -
param c = a + a;            //   binary + and -
param d = c / a;            //   binary *, /, min, max on real, imag
param x = 1.0 / 2.0;        // now works
param y = 3.0 + 4.0i;       // now works
```

- Generated C now uses hex float syntax for 'param' floating point values
  - to avoid potential for rounding error in decimal-binary conversions
- INFINITY and NAN are now 'param' values

param real: Impact, Next Steps

Impact:

- Improved ease-of-use
- Enabled work on library improvements for floating point attributes

Next Steps:

- Add support for ‘*’ and ‘/’ on 'param' 'complex'
- Enable 'param' evaluation of select Math functions
  - pow(), exp(), sin(), ...
- Continue to work towards satisfying IEEE 754 support

'forall' vs. '[]' loops

forall vs. [ ]: Background

- '[ ]' was considered a syntactic convenience for 'forall'
  - both forms required parallel iterators
  - falling back on a serial implementation was not supported

```
// data-parallel statements required parallel myIter()
forall idx in myIter() do writeln(idx);
[ idx in myIter() ] writeln(idx);

// ditto data-parallel expressions
process(forall x in myIter() do x + 1);
process([ x in myIter() ] x + 1);
```

1.18: same behavior of forall and [ ] syntax

forall vs. [ ]: This Effort

- **forall**-loop now ensures that parallel iterator(s) are invoked
  - either standalone (non-zippered loops only) or leader+follower(s) [1]
  - compiler generates an error if they are not available

```
process(forall x in myIter() do x + 1);
```

1.19: 'forall' requires parallel iterators

- **[ ]**-loop falls back on serial version(s) when parallel versions are not available

```
[ x in serialIter() ] writeln(x);
```

1.19: '[ ]' falls back on serial iterators

- Mnemonic: When I say 'forall', I mean "parallel"

forall vs. [ ]: Zippered [ ]-loops

A zippered [ ]-loop:

- runs in parallel only when ALL iterable expressions support parallelism

```
// arrays, domains usually support parallelism
[ tup in zip(MyArray, MyDomain) ] process(tup);
```

- otherwise runs serially

```
// when serialIter() has no parallel versions
var A = [ (i,a) in zip(serialIter(), MyArray) ] i*a;
[ tup in zip(MyDomain, serialIter()) ] process(tup);
```

forall vs. [ ]: reduce and promoted expressions

- Reduce- and promoted expressions allow fallback on serial iteration, as before

```
var sum = + reduce myIter();     // sum reduction
var radii = myIter() / 6.28;     // promotion of myIter()
```

  - mnemonic: no 'forall' keyword → no parallelism required

- This release: compiler reports an error when:
  - a parallel iterator is available, and
  - there is an error while resolving it, e.g., a typo

    (cf. used to resort to serial iteration instead)

upon error in parallel myIter():
1.18: switch to serial
1.19: report to user

forall vs. [ ]: Impact

- Exposed cases where serial iteration was unintentional
  - because errors in parallel iterators were not reported to user

- Simpler code for the "OK to resort to serial" pattern

```
if <MyData supports parallelism> then
  [ elm in MyData ] process(elm);
else
  for elm in MyData do process(elm);
```

1.18: if-then-else was needed

1.19: one [ ]-loop suffices

forall vs. [ ]: Status

- The choice of parallel vs. serial iteration is correct in most cases
- However, the serial iterator is still chosen incorrectly in some cases
  - when initializing an inferred-type variable via a forall expression:

```
// 'forall' uses serial instead of parallel version of myParIter()
var A = forall i in myParIter() do idx;
```

  - when using a forall expression as the iterand in a []-loop:

```
// 'forall' runs serially instead of reporting "error:  parallel version of serialIter() is not
// available"
[ i in (forall j in serialIter() do j) ] process(i,j);
```

forall vs. [ ]: Next Steps

- Resolve remaining incorrect cases
- Improve language and compiler support for parallel iterators
  - better ways to declare parallel versions of an iterator
    - require all versions to be declared together in the source code?
    - implement as methods on an object?
  - avoid resolving the serial version when only the parallel versions are used?

```
// ex. to flag attempts of serial execution with a compiler error
iter amIparallel() { compilerError("must run in parallel"); }
iter amIparallel() /* a parallel version */ { … generate parallelism … }
```

forall vs. [ ]: Iterator Forwarders

- Language support for "iterator forwarders"
  - "do this and that, then redirect to iterator X"

```
// pseudo-code: iterate() forwards to iterateHelp()
iter RandomStream.iterate(D: domain, type resultType) {
  const start = _count; _count += D.numIndices; ...
  /* then go to */ iterateHelp(resultType, D, seed, start);
}
```

  - allow forwarding to apply to parallel versions, too?

```
// allow a parallel version of iterateHelp() to execute here?
forall myRandomStream.iterate(D,int) do ...;
```

Shape / Index Preservation

Shape Preservation: Background

- Recent releases have improved the preservation of shapes/indices:

```
var A, B: [1..3, 1..3] real;

var C = A + B;              // C's domain used to be {1..9}, is now {1..3, 1..3}
var D = [a in A] a**3;      // ditto for D
```

- Scans and range expressions did not benefit from these improvements:

```
var S = + scan A;           // S.domain was {1..9}

proc f(i: int) return i+7;
const R = -1..7;
var G = f(R);               // G.domain was {1..9}
```

Shape Preservation: This Effort, Impact

This Effort: Extended shape/index preservation to scans and ranges

- Also enabled parallelism, see *Performance and Benchmarks* slides

Impact: Scans and range-based expressions now behave much more intuitively

```
var A: [1..3, 1..3] real;
var S = + scan A;              // S.domain is now {1..3, 1..3}

proc f(i: int) return i+7;
const R = -1..7;
var G = f(R);                  // G.domain is now {-1..7}
```

Shape Preservation: Next Steps

- Add a way to create shape-ful iterators?

```
var A: [1..3, 1..3] real;

proc myIter() {     // 'myIter' yields all values in a {1..3,1..3}-shaped loop
  for a in A do
    yield process(a);
}

var B = myIter() + 3;
```

- We want to infer, or allow the user to declare, that:
  - myIter() is shapeful, such that B.domain is {1..3, 1..3}
  - B can be computed in parallel, given that data parallelism is available over A

Numeric Literals with Underscores

Underscores in Numeric Literals

Background: Numbers with many digits are difficult to visually parse

```
const n = 400000000000;    // "Four hundred billion" or "Four trillion"?
const x = 0.000000003;     // "3*10-9" or "3*10-10"?
```

This Effort: Allowed underscores in numeric constants

- Now allowed for all numeric types

Impact: Numbers with many digits are easier to read

```
const n = 400_000_000_000;
const x = 0.000_000_003;
```

String to Numeric Casts

String Casts: Background, This Effort

Background: String-to-numeric casts didn't support the same formats as literals

- Only supported casts from strings in base-10 to integer

```
var n = "0xff": int;        // error: bad cast from string '0xff' to int(64)
```

- Underscore separators in numbers were not supported

```
var x = "10_000": int;    // error: bad cast from string '10_000' to int(64)
```

This Effort: Improved string cast support for numeric types

- Allow integral casts from binary, octal and hexadecimal strings
- Allow underscores in integer and floating point strings

String Casts: Impact, Status

Impact: Strings cast to numeric types can resemble literal values more closely

```
"0xfedc": int == 0xfedc
"0b1010": int == 0b1010
"1_234.56e7i": imag == 1_234.56e7i
```

- int(64) cast performance improved, other int sizes got slightly slower

Status: String-to-numeric casts support the same formats as literals

Default Assignment and Equality Operators for Records

record ==: Background

- Chapel used to allow some operations between records of different types
  - assignment with =, comparison with == or !=
- This resulted in surprising behavior in some cases

```
record R { var x; }
var r64: R(int(64));
var r32: R(int(32));
r64 = r32;  // allowed in 1.18
r32 = r64;  // compilation error in 1.18
```

record ==: This Effort, Impact

This Effort: Compiler-generated =, ==, != now require records of the same type

- Reduces compiler complexity
- Simplifies the language design
- Users can still get old behavior by creating custom operator overloads, e.g:

```
proc =(ref lhs: R, rhs: R) {
  lhs.x = rhs.x;
}
```

Impact: Default behavior for records is simplified

New Reserved Words

Reserved Words: Background

- Primitive types had historically not been reserved words
  - Occasionally lead to confusing code and strange errors

```
var int = 1;
int = 2;
var x: int;  // error: invalid type specification
```

- Other languages make such basic type names reserved words

Reserved Words: This Effort, Impact

This Effort: Core built-in types and values are now reserved

- Redefining these would be more confusing than useful

bool	true	false
real	imag	complex
int	uint	locale
string	this	

Impact: Error messages are improved

```
var int = 1;  // error: attempt to redefine reserved type 'int'
int = 2;
var x: int;
```

For More Information

For a more complete list of language-related changes in the 1.19 release, refer to the following sections of the [CHANGES.md]() file:

- Syntactic/Naming Changes
- Semantic Changes
- New Features
- Feature Improvements
- Deprecated and Removed Features
- Standard Modules / Library
- Error Messages / Semantic Checks

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.

THANK YOU

QUESTIONS?

chapel_info@cray.com

@ChapelLanguage

chapel-lang.org

cray.com

@cray_inc

linkedin.com/company/cray-inc-