Language Improvements

Chapel Team, Cray Inc.
Chapel version 1.16
October 5, 2017

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

Error Handling

Error Handling: Background

- **1.15 contained a draft implementation of error handling**
  - Basic functionality with ‘try’, ‘catch’, ‘throw’, etc.
  - Default and strict modes
- **Feedback was positive**
- **But implementation was known to be incomplete**
  - Could not be used with tasks, ‘on’ statements
  - Error handling modes were too coarse-grained
  - No ‘Error’ hierarchy defined in the standard library

Error Handling: This Effort

- **Implement incomplete features**
  - Enable error handling across tasks and locales
  - Make error modes more fine-grained
  - Create a ‘SystemError’ hierarchy
  - Provide a primer, improve documentation
- **Solidify the implementation**
  - Use error handling in the standard library
  - Improve compile-time checking for method overrides
  - Close bugs and memory leaks

Error Handling: Multilocale

- Errors from ‘on’ statements can now be handled

```
try {
  var x: int;
  on Locales[1] {
    x = throwingCall();
  }
  return x + 1;
} catch {
  writeln("caught an error from locale 1");
  return 0;
}
```

Error Handling: Parallelism, TaskErrors

- **'TaskErrors' help to aggregate errors across tasks**
  - 'Error' subtype that collects errors from tasks for central handling
  - Only thrown if there are one or more errors from the tasks
  - Can be iterated on, filtered for different kinds of errors

```
try {
  ...
} catch errors: TaskErrors {
  for e in errors {
    writeln("Caught task error e ", e.message());
  }
}
```

Error Handling: Parallelism, ‘begin’

- **Errors can be thrown from a ‘begin’ statement**
  - Waiting ‘sync’ statement collects them in a ‘TaskErrors’
    - Note: ‘sync’ statements are always considered to throw

```
try {
  sync {
    begin canThrow(0);
    begin canThrow(1);
  }
} catch errors: TaskErrors {
  ...
}
```

Error Handling: Parallelism, ‘cobegin’/‘coforall’

- ‘cobegin’ blocks and ‘coforall’/‘forall’ loops can also throw
  - Errors will be stored in a ‘TaskErrors’, even if only one task is run

```
try {
  cobegin {
    canThrow(0);
    canThrow(1);
  }
} catch e: TaskErrors {
  ...
}
```

```
try {
  coforall i in 1..2 {
    throw new DemoError();
  }
} catch e: TaskErrors {
  ...
}
```

Error Handling: Nested Parallelism

- **Nested loops or tasks do not produce nested 'TaskErrors'**
  - All errors are flattened to a single level

```
try {
  forall i in 1..2 {
    forall j in 1..2 {
      throw new DemoError();
    }
  }
} catch errors: TaskErrors {
  ...
}
```

Error Handling: Error Modes, intro

- **Error handling modes are now set per module**
  - '--strict-errors' compiler flag has been removed

1. **Fatal mode**
   - No error handling is required, but implicit halts will be inserted
   - Same as Default mode from 1.15
2. **Relaxed mode**
   - Error handling is required in non-throwing functions only
   - New mode, not present in 1.15
3. **Strict mode**
   - Requires that all throwing calls be marked by 'try' or 'try!'
   - Same as Strict mode from 1.15

Error Handling: Error Modes, intro

This chart shows the behavior of each error-handling mode when an unhandled throw is in…

…a non-throwing function

…a throwing function

	Program halts	**Compiler generates an error**
Errors are thrown upward	Fatal mode	Relaxed mode
Compiler generates an error	X	Strict mode

Error Handling: Error Modes, implicit modules

- **Implicit modules use Fatal mode by default**

```
// implicitModule.chpl

proc doesNotThrow() {
  // the program will halt if an error is thrown
  thisCallMayThrow();
}

proc doesThrow() throws {
  // if an error is thrown, it will be thrown upwards
  thisCallMayThrow();
}
```

Error Handling: Error Modes, explicit modules

- **Explicit modules use Relaxed mode by default**

```
module E {
  proc doesNotThrow() {
    // the error must be handled by a try! or a try with a catch all
    try! thisCallMayThrow();
  }

  proc doesThrow() throws {
    thisCallMayThrow(); // still throws any errors upward implicitly
  }
}
```

- **Default for explicit modules can be changed to Fatal mode**
  - '--permit-unhandled-module-errors' compiler flag

Error Handling: Error Modes, ‘prototype’

- **‘prototype’ modules are new in this release**
  - Allows explicit modules to be used more casually
  - Currently used for error handling, looking to expand its utility

```
prototype module P { ... }
```

- **Prototype modules use Fatal mode by default**

```
prototype module NP {
  proc doesNotThrow() {
    thisCallMayThrow(); // this will halt if an error is thrown
  }

  proc doesThrow() throws {
    thisCallMayThrow(); // still throws an error upward implicitly
  }
}
```

Error Handling: Error Modes, strict

- **Strict mode can be enabled for a module**
  - Uses a pragma for now
  - All throwing calls must be marked with ‘try’…
    …even if no handling occurs in the function

```
pragma "error mode strict"
module S {
  proc doesNotThrow() {
    // the program will halt if an error is thrown
    try! thisCallMayThrow(); // explicitly marked
  }
  proc doesThrow() throws {
    // if an error is thrown, it will be thrown upwards
    try thisCallMayThrow(); // explicitly marked
  }
}
```

Error Handling: 'try' expressions

- **'try'/'try!' expressions are available for cleaner idioms**
  - Easily marks throwing calls in Strict mode

```
proc idiomOne() throws {
  var x = try intOrThrow(0); // throws errors upwards
  var y = try intOrThrow(1);
  return x + y;
}

proc idiomTwo() {
  return try! idiomOne(); // halts on error
}
```

Error Handling: 'SystemError' hierarchy

- **Contains class types for common system error codes**
  - i.e. 'ChildProcessError' for ECHILD
  - Subtypes include 'ConnectionError', 'FileNotFoundError', etc.
    - Based on Python's 'OSError'
- **Function helps generate 'SystemError' from error codes**
  - Note: error codes are represented by 'syserr'

```
use SysError;
proc callAndConvert() throws {
  var e:syserr;
  someOutErrorCall(e);
  if e then
    throw new SystemError.fromSyserr(e);
}
```

Error Handling: Standard Modules

- **Updated many modules to throw 'SystemError'**
  - Serves as an important demonstration of error handling
  - Removes halts from typical use
- **Modules updated:**
  - IO
  - FileSystem
  - Path
  - Spawn
  - Buffers
  - Regexp
  - HDFS

Error Handling: Status & Next Steps

Status:

- Stable, ready for use in production
- Soliciting community feedback, particularly on error modes

Next Steps:

- Enable throwing from initializers, non-inlined iterators
- Consider throwing from deinitializers, ‘defer’ statements
- Investigate if we can reduce number of error modes, w/ user feedback
  - May include changing Strict mode to a warning control option
- Explore stack tracing interaction
- Allow captured Errors to be saved and not deleted
- Modify the runtime to use error handling
- Update BigInt to throw ‘SystemError’ instead of halting

Defer Statements

Defer: Background

- **Sometimes a variable represents a resource**
  - e.g. a held lock, an open file, allocated memory
- **When should the resource be released?**
  - Generally, just before the variable goes out of scope
- **Currently, Chapel supports an RAII pattern with records**

```
// initializing a record variable acquires the resource
var myfile = open("myfile.txt", iomode.r);
// resource is released on block exit through record destructor
return;
```

- **But using a record might be too much effort**
  - challenging when release function needs more args/returns a value

Defer: This Effort

- **Add a 'defer' statement to enable general resource release**
- **A 'defer' specifies a cleanup action for an enclosing block**
- **Cleanup actions run for *any* block exit**
  - through regular exit
  - function return
  - error handling
  - 'break' and 'continue' within loops

Defer: Ugly Example

```
proc f(out releaseError:int) throws {
  var resource = allocateResource();
  var myInstance = new MyClass();
  if !setupResource(resource) {
    delete myInstance; // free resources if setup fails
    releaseError = releaseResource(resource);
    return;
  }
  try {
    throwingFunction();
  } catch e {
    delete myInstance; // free resources if we're throwing an error upwards
    releaseError = releaseResource(resource);
    throw e;
  }
  delete myInstance; // free resources upon normal return
  releaseError = releaseResource(resource);
}
```

Defer: Ugly Example Rewritten to Use 'Defer'

```
proc f(out releaseError:int) throws {
  var resource = allocateResource();
  var myInstance = new MyClass();
  defer {
    // free resources regardless of which of the three ways we might return
    delete myInstance;
    releaseError = releaseResource(resource);
  }
  if !setupResource(resource) {
    return;
  }
  try throwingFunction();
}
```

Defer: Impact and Next Steps

Impact:

- Easier to express complex resource management
- Enabled cleaner resource reclamation in the compiler

Next Steps:

- Decide if errors can be thrown from defer blocks

Reduce Intents Preserve Initial Value

Reduce Intents: Background

- **Reduce intents support reductions alongside other work**

```
var sum = 0, maxval = min(int);
forall a in A with (+ reduce sum, max reduce maxval) {
  sum += a;                    // accumulate into sum reduction variable
  maxval = max(maxval, a);     // accumulate into max reduction variable
  a = a * 2;                   // double each element
}
```

- **In 1.15, final reduction ignored outer variable's value**

```
var sum = 0;
for i in 1..3 do
  forall a in A[i] with (+ reduce sum) do
    sum += a;
writeln(sum); // in 1.15, only 'A[3]' was summed up
```

'forall' over 'A[3]' overwrote previously-computed sums of 'A[1]' and 'A[2]' elements

- **Users prefer incorporating the initial value**
  - matches the behavior of OpenMP reduction clause

Reduce Intents: This Effort and Next Steps

This Effort: include the variable's initial value, too

```
var sum = 0;
for i in 1..3 do
  forall a in A[i] with (+ reduce sum) do sum += a;
writeln(sum); // now includes 'A[1]', 'A[2]' ,and 'A[3]' sums
```

includes the previous sums

- note: reduce-expressions are not affected
  - the initial value is always the reduction op's identity

Next Steps: explore a way to "just get the reduction result"

- relieve user from concerns about initial value, result type
  - want the result type to be inferred automatically

```
var result: ???; // syntax tbd; type to be determined by 'MyLibraryReduction'
forall a in A with (MyLibraryReduction reduce result) do
  result reduce= a;
writeln(result);
```

Enabling 2-way Return Intent Overloads

Return Intent: Background and This Effort

Background: Chapel supports return intent overloads

```
proc access() : int { ... }                  // value version
proc access() : int ref { ... }              // ref version
proc access() : int const ref { ... }        // const ref version
var x = access(); // resolves to value version
ref r = access(); // resolves to ref version
const ref r = access(); // resolves to const ref version
```

These only worked if one function had ‘ref’ return intent

This Effort: Support the case without a ‘ref’ return overload

```
proc access() : int { ... }                  // value version
proc access() : int const ref { ... }        // const ref version
```

Return Intent: Impact

- **Return intent overloads are more flexible**
- **Optimization used for arrays can apply to read-only cases**
  - supply a value overload for small types like ‘int’
    - so that, for example, compiler can add copies of the value across locales
  - ‘const ref’ overload is still important for correct behavior
    - e.g. ‘A[i].locale’ should return where the element is stored, not always ‘here’
  - in 1.15, could not do that because had to provide ‘ref’ overload
    - ‘ref’ is illegal for returning read-only data

Returning and Yielding ‘void’

Returning and Yielding void

Background: Returning/Yielding ‘void’ was not well defined

- Non-returning functions could be assigned to ‘void’ variables
  - The same as if they returned the value ‘_void’
- Iterators yielding ‘_void’ caused compiler errors
- Iterators with no ‘yield’ statements caused compiler errors

This Effort: Better define ‘void’ returns and yields

- Functions that do not return cannot be assigned to variables
- Functions that return the value ‘_void’ can be assigned to variables
- Iterators that do not ‘yield’:
  - execute the iterator body
  - …but not the body of the calling loop
- Iterators that ‘yield’ the value ‘_void’:
  - execute the loop body once per yield
  - have a ‘void’ index variable in the calling loop

Returning and Yielding void

Impact: Returning void is better defined

- Assigning ‘void’ function to a variable is allowed if it returned ‘_void’

```
proc returnVoid() {
  writeln(“explicit void”);
  return _void;
}
// v.type is void
var v = returnVoid();
```

- A function that doesn't return can no longer be assigned to a variable

```
proc noReturn() {
  writeln(“no return”);
}
// now a compiler error
var v = noReturn();
```

Returning and Yielding void

Impact: Yielding void is better defined

- Iterators can yield 'void' values

```
iter yieldVoids() {
  for i in 1..3 do
    yield _void;
}
for i in yieldVoids() do
  writeln("Hello"); // Printed 3 times
```

- Iterators can execute without yielding anything

```
iter yieldNothing() {
  writeln("In yieldNothing"); // Printed
}
for i in yieldNothing() do
  writeln("Hello"); // Not printed
```

Returning and Yielding void

Next Steps:

- Determine better name for void value than '_void'

Conditional Local Statement

Conditional Local Statement

Background: local statements squash communication overhead

- Runtime halts if communication is found

```
local Foo.updateElements();
```

- Conditional use of local statements was verbose

```
if Foo.locale == here then
  local Foo.updateElements();
else
  Foo.updateElements();
```

This Effort: Support local statements with runtime conditions

- No optimization if condition is false

```
local Foo.locale == here do Foo.updateElements();
```

- Single-statement bodies must use ‘do’:

```
local foo();       // OK in 1.15, illegal in 1.16
local do foo();    // New in 1.16
```

- Compiler expands such local statements into the conditional above

Conditional Local Statement

Impact: Improved elegance of local statements

- Syntax change required minor updates to benchmarks/internals

Next Steps: Continue effort towards data-centric locality

- Local statement is unwieldly and limits scope
- Ideally local statements will be unnecessary in the future

Initializers

Initializers: Background

- **Chapel's traditional constructor story was naïve**
  - Became increasingly clear as users/developers relied on OOP more
  - Lacked a good copy constructor / initializer story
- **Have been developing 'initializers' as a replacement**
  - Consist of two phases:
    - phase 1: constrained initialization of fields
    - phase 2: general computation
  - Phases separated by call to one of:
    - super.init, for initialization of inherited fields (if any)
    - this.init, for common operations on the type (including field initialization)
  - Design being managed in CHIP 10
    - See also the 1.13 release notes on constructors
  - Constructors will be deprecated once initializers are complete

Initializers: Background

- **Last release: extended support for initializers**
  - Significant support for initializers on non-generic types
    - Good coverage with semantic checks
  - Preliminary support for initializers for generic types
    - Concrete base classes derived from body of user-defined initializers
    - No support for derived classes
    - No support for generic records
    - No support for parameterized type declarations
  - No support for compiler generated initializers
  - Further details in the 1.15 release notes on initializers

Initializers: Summary of this Effort

- **Initializers for non-generic class/record now robust**
  - Many bug fixes
  - Improved semantic checks
- **Initializers for generic class/record implemented**
  - However some important use-cases are incomplete
- **Preliminary support for compiler-generated initializers**
  - Can be enabled for user-defined classes with a developer flag
  - Compiler-generated constructors remain the default

Generics and compiler-generated initializers are covered in more depth in the following slides

Generic Initializers

Generic Initializers: Background

- **Constructors support generic types**
  - Compiler generates a type constructor for every generic type
    - One argument for every generic field in field order, from base to derived

```
class Foo {
  var x;
  type t = bool;
  proc Foo(x, type t) {
    this.x = x;
  }
}
```

Every constructor requires a formal per generic field with the same name

Can't assign to 'type' or 'param' fields in the body, their value is taken from the argument directly

  - Types instantiated via 'new' calls or via type constructor

```
var x: MyType(…); // sets 'x' to default value for generated type
// set y to default value of generated type, then assign to result of 'new' call
var y: MyType(…) = new MyType(…);
var z = new MyType(…); // sets z to result of 'new' call
```

Generic Initializers: Background

- **Last release added initializer support for generic classes**
  - Allowed ‘type’, ‘param’, and generic ‘var’/‘const’ fields
    - Type instantiation based on Phase 1 operations, not arguments
    - Omitted initialization allowed when declared type or initial value provided

```
class Foo {
  var x;  // can't omit this field
  type t = bool;  // allowed to omit this one
  proc init(xVal) {
    x = xVal;
    // omission of 't' means it is set to 'bool'
    super.init(); // instantiated type is 'Foo(xVal.type, bool)'
  }
}
```

No requirement placed on arguments

Generic Initializers: Background

- **Last release only supported generic classes via 'new' calls**

```
class Baz {
  var x;

  proc init(xVal) {
    x = xVal;
    super.init();
  }
}

var z = new Baz(11); // worked for generic classes but not generic records
```

Generic Initializers: Background

- **Limitations from last release**
  - Generic records were not well supported
  - Inheritance when the parent type is generic was not well supported
  - Issue with ‘this.init()’ calls to other initializers defined on the type
    - Initializer expected to know generic instantiation by that call
      …but the initializer it called was responsible for determining it

Generic Initializers: Background

- **Last release, initializers suppressed type constructors**
  - Types defined solely by Phase 1 operations in resolved initializers

```
class Foo {
  type t;
  param p;

  proc init(…) {
    …
  }
}
```

```
class Bar {
  var x;

  proc init(xVal) {
    x = xVal;
    super.init();
  }
}
```

```
var x: Foo(int, 4);                          // failed last release
var y: Bar(bool) = new Bar(false);           // failed last release
```

Generic Initializers: This Effort

- **Generate a type constructor for a generic with initializers**
  - One argument per generic field (like generics with constructors)
    - ‘type’ fields require a ‘type’ argument
    - ‘param’ fields require a ‘param’ argument
    - Generic ‘var’/‘const’ fields require a ‘type’ argument

```
class Foo {
  type t;
  param p;

  proc init(…) {
    …
  }
}
```

```
class Bar {
  var x;

  proc init(xVal) {
    x = xVal;
    super.init();
  }
}
```

```
var x: Foo(int, 4);                          // now works!
var y: Bar(bool) = new Bar(false);           // now works!
```

Generic Initializers: This Effort

- **Added support for generic records**
- **Default value for the record now requires an initializer with an argument per generic field**

```
record MyRec {
  param field;
  …
}
```

```
proc MyRec.init(param p) {
  field = p;
  super.init();
}
```

```
var x: MyRec(5);  // transformed to 'var x: MyRec(5) = new MyRec(5);'
```

- Was supported by default constructors before
  - Explicit initializers prevent creation of default initializer (or constructor)
- Of course, additional initializers permitted

Generic Initializers: This Effort

- **Added support for inheriting from a generic parent class**
  - Open issue with using an inherited type field as a field's type

```
class Foo {
  type t;
  …
}
```

```
class Bar: Foo {
  var x: t; // Fails with initializers, t is not defined yet
  …
}
```

    - No current workaround

- **Fixed bug with 'this.init()' calls on generic types**
  - Among various other minor bugs

- **Extended verification checks to apply to generic types**

Generic Type Declarations: Status

- **Overall, initializers on generic types are greatly improved**
  - Initializers on generic types more powerful than constructors were
- **Current state leaves some potential pitfalls for the user**
  - Due to new patterns that couldn't be written with constructors
    - These new patterns seem desirable
    - The potential confusion might outweigh the benefits, though

  - The following slides cover these cases in more detail

Generic Type Declarations: Status

- **Possible to create generic objects w/ limited functionality**

```
class Foo {
  type t;
  param p;

  proc init() {
    t = real;
    p = 3;
    super.init();
  }
}
```

```
var x: Foo(int, 4);   // Won't error: 'x' is nil with type 'Foo(int, 4)'.  Should it?

x = new Foo();        // Error: init generates an instance of 'Foo(real, 3)'
```

- Allows type creator to place limits on the type instantiations available

Generic Record Type Declaration: Status

- **Implications: can generate a type that cannot be declared**

```
record MyRec {
  param p;
  var v: int;
  proc init(vVal: int) {     // the only initializer doesn't have a param arg
    p = 5;
    v = vVal;
    super.init();
  }
}
```

  - Type is valid but default value requires 'proc init(param p);'

```
var rec1: MyRec(5);                       // Does not work
```

  - Compiler continues to view this as default init followed by assign

```
var rec2: MyRec(5) = new MyRec(3);        // Does not work currently
```

Generic Record Type Declaration: Status

- **Implications: initializer may generate ‘surprising’ type**
  - This will also make the record more difficult to use

```
record MyRec {
  param field;

  proc init(param p) {
    field = p + 2;
    super.init();
  }
}
var rec1: MyRec(5); // Fails: default value has type 'MyRec(7)'

var rec2: MyRec(5) = new MyRec(3); // works! (is this too surprising?)
```

Default Initializers

Default Initializers: Background

- **Compiler generates a default constructor per type**
  - Unless an explicit initializer is provided
- **Available when no constructor/initializers are defined**
- **Every user constructor makes a call to this constructor**
  - Compiler inserts it automatically as the first line in the body
  - Complicates generated code
  - Adds overhead during object construction

Default Initializers: Background

- **Default Constructor Semantics:**
  - Has a formal per field with the same name as the field
  - Initial values used as the default value for argument, when present
    - Uses default value of type as argument default value otherwise
    - No type or initial value means the argument is required

```
class Foo {
  var x = 5;
  var y: bool;
}
```

Compiler-generated default constructor

```
proc init(x=5, y: bool = false) {
  this.x = x;
  this.y = y;
  super.init();
}
```

```
var c1 = new Foo();   // c1.x = 5, c1.y = false
var c2 = new Foo(6);  // c2.x = 6, c2.y = false
```

  - Handles memory allocation

Default Initializers: This Effort

- **Added support for generating class default initializers**
  - Semantics are similar to default constructors
    - Difference: arguments for inherited fields come after derived ones
      - Reflects order of initialization – inherited fields initialized through ‘super.init()’ call

  - Types with defined constructors still generate a default constructor
    - As do types with ‘initialize()’ method

  - Types that define an initializer will not get a default initializer

  - Types with no constructor/initializer get a default constructor
    - New behavior is enabled by a developer flag
    - And then only for classes in user-defined modules

Default Initializers: Status

- **Can generate default initializers for many user classes**
  - Classes with proper field dependencies, e.g.:

```
class Foo {
  var x: int;
  var y = x + 2; // y depends on x, which was previously defined
}
```

    - Fields that depend on later fields may lead to an unclear error message, i.e.

```
class Foo {
  var x = y + 2; // x cannot depend on y, but the error message is unclear
  var y: int;
}
```

  - Nested non-generic classes
  - Some generics
  - And the compiler's internal support for first class functions

Initializers: Overall Status and Next Steps

Initializers: Status

- **Converting constructors to initializers in progress**
  - Most nightly tests have been converted
  - Work on Internal/Standard modules has begun

- **Today, 2,135 standard tests include non-library types**
  - 63 of these include constructors
    - Primarily for continued coverage (32 tests)
    - A few were overlooked accidentally (16 tests)
    - A few types inherit from internal classes with constructors (10 tests)
    - Rest are ‘noinit’ tests or due to a bug with nested generic types (5 tests)
  - 58 include the complementary ‘initialize()’ method
  - 369 include ‘init()’ methods
  - Rest define no constructor or initializer

Initializers: Status

- **225 of 1,645 tests with no constructor/initializer fail with default initializers enabled (13.7%)**
  - Lots of small failure categories, most of which are diagnosed
  - Largest are:
    - Inheritance from library types (65 tests, 28.9% of failures)
    - Sync fields without explicit initial values cause deadlocks (28 tests, 12.4%)

```
class Foo {
  var s$: sync int;
}
```

Initializers: Next Steps

- **Specific default initializer next steps, in order:**
  1. Resolve remaining test cases with class errors, except inheritance from library types
     - Will get resolved when we apply default initializers to library types
  2. Have compiler generate default initializers for records defined in user code
  3. Have compiler generate default initializers for internal/library classes and records
  4. Enable as the default behavior for all types

Initializers: Next Steps

- **Other initializer next steps**
  - Fix bugs
    - Nested types when at least one of the involved types is generic
    - Utilizing an inherited ‘type’ field
    - Generic instantiation when generic fields initialized in conditionals
    - ‘param’ fields with omitted initialization when of non-default ‘int’ type
  - Deprecate constructors
  - Revisit potential pitfalls to ensure we want that behavior

Initializers: Next Steps

- **Evaluate design decisions as we convert existing code**
  - Syntax for division between phases 1 and 2
  - What should the default phase be?
  - Could the distinction between phases be blurred in common cases?
    - e.g., support params / types in phase 2 since evaluated at compile-time?
  - Should ‘const’ fields be re-assignable in phase 2?

Initializers: Next Steps

- **Other design decisions to revisit**
  - Generation of default initializers (currently squashed by user's 'init()')
    - Should users be able to opt-into retaining compiler's default 'init()'?
  - Importance of possible optimizations
  - Deprecation of the 'initialize()' method
    - Or should we support it, but with a different name?

Other Language Improvements

Other Language Improvements

- **Removed support for '~' operator on bools (bitwise NOT)**
  - '!' (logical NOT) still supported
- **Improvements to forwarding a field**
  - Include inherited methods when forwarding
  - Fixed an issue with generic instantiation of forwarded methods
  - Iterators can now be forwarded
- **Added a requirement that 'deinit()' must have parentheses**

Documentation Improvements

Documentation Improvements

- **Language Spec Improvements:**
  - Added a new ‘methods’ chapter with refreshed / reorganized content
  - Improved definition of records
  - Documented that ‘=‘ is overloadable
- **Online Doc Improvements:**
  - Added missing online docs for missing range queries
    - .low, .high, .stride, .alignment, .aligned
  - Added missing documentation for reindex(), localSlice()

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