Ongoing Efforts

Chapel Team, Cray Inc.
Chapel version 1.17
April 5, 2018

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

- **Delete-Free Chapel**
  - Purpose of this effort
  - General Goal
  - Strawman Design
  - Progress towards Prototype
  - Open Questions
- **Function Hijacking**
- **Open Fabrics Interface (‘ofi’) Communication Layer**

Delete-Free Chapel

Delete-Free: Outline

- Purpose of this effort
- General Goal
- Strawman Design
- Progress towards Prototype
- Open Questions

Purpose of this effort

Memory Management Strategies Scorecard

Garbage Collection	'delete'
+ safety guarantees + eliminates memory leaks + eliminates double-delete + eliminates use-after-free	– more errors possible – failure to delete results in leaks – double-delete possible – use-after-free possible
+ ease-of-use + no need to write 'delete'	– more burden on programmer – think about 'delete'
– implementation challenges due to distributed memory & parallelism	+ simpler implementation
– performance challenges – stop-the-world interrupts program – concurrent collectors add overhead – scalability may prove difficult	+ predictable, scalable performance

- **Based on these tradeoffs, Chapel started with 'delete'**

First Step: Owned and Shared

- **General-purpose wrapper records help avoid 'delete'**
- **Owned: uses a single-owner pattern to manage lifetime**
  - contained class instance deleted when 'Owned' goes out of scope
  - assignment and copy initialization are destructive ownership transfers
- **Shared: uses reference-counting to manage lifetime**
  - contained class instance deleted when all 'Shared' copies destroyed
  - assignment and copy initialization share ownership
- **Introduced in version 1.15**

Owned and Shared: Safety Properties

- **Are memory leaks still possible? Yes.**

```
var x = new MyClass();
// leak: x never deleted

var y = new Owned(new MyClass());
y.release();
// leak: y's instance never deleted
```

Owned and Shared: Safety Properties

- **Is use-after-free possible? Yes.**
  - Storing a borrow from a local variable into a global:

```
var global: MyClass;
{ // bad borrow
  var y = new Owned(new MyClass());
  global = y.borrow();
  // instance deleted here
}
...global... // use-after-free!
```

  - Using an unmanaged pointer after it is managed:

```
{
  global = new MyClass();
  var z = new Owned(global);
  // instance deleted here
}
...global... // use-after-free!
```

Owned and Shared: Safety Properties

- **Is use-after-free possible? Yes.**
  - Invalidating an Owned while a borrow exists:

```
var a = new Owned(new MyClass());
var b = a.borrow();
a.clear(); // deletes a's instance!
a.retain(new MyClass()); // deletes a's instance!
...b... // use-after-free!
```

  - Shrinking an array while a borrow to an element exists:

```
var D = {1..1};
var A: [D] Owned(MyClass);
A[1] = new Owned(new MyClass());
var b = A[1].borrow();
D = {1..0}; // Assigning to domain resizes A and destroyes A[1]
...b... // use-after-free!
```

Owned and Shared Scorecard

- **Owned and Shared remove the need to write 'delete' but do not address memory safety**

Owned and Shared
– not much safer than 'delete' – double-delete possible – use-after-free possible
+ no need to write 'delete' – have to mark variables/fields as Owned/Shared
+ manageable implementation
+ low impact on execution-time program performance

Background: Rust

- **Rust's approach prevents memory errors at compile time**
  - programs that might have a use-after-free result in compilation error
  - its *borrow checker* is the component raising these errors
- **Rust's approach also prevents race conditions**
  - since race conditions can introduce memory errors
- **Rust programmers can also write *unsafe* code**
  - provides a way to opt out of the above checking
  - expectation is that unsafe code is carefully inspected

Motivating Question

- **Can Chapel include something Rust-like?**
  - compile-time detection of use-after-free?
- **The Big Issue: Complete Checking and Race Conditions**
  - recall that a race condition can introduce a use-after-free error
  - For example:

```
proc test() {
  var myOwned = new Owned(new MyClass());
  var b = myOwned.borrow();
  cobegin with (ref myOwned) {
    { myOwned.clear(); }     // deletes instance
    { writeln(b); }          // races to use instance before delete
  }
}
```

Complete Checking and Race Conditions

- **Should Chapel rule out race conditions at compile time?**
- **A worthy goal, but the Rust strategy doesn't fit Chapel**
  - only one mutable reference to an object can exist at a time
  - if a mutable reference exists, no const references to that object
- **Such a strategy in Chapel would make these illegal:**

```
forall a in A { a = 1; }
forall i in 1..n { A[i] = i; }
forall i in 1..n { B[permutation(i)] = A[i]; }
```

- **Could a different strategy detect these race conditions?**
  - Maybe, but it would be difficult
  - Can the compiler prove that 'permutation' is a permutation?
  - If not, how would that be communicated to the compiler?

General Goal

General Goal

- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**

Proposal: Lifetime Checking
+ helps with safety + eliminates many memory leaks + eliminates many double-delete + eliminates many use-after-free – but doesn't catch all cases
+ no need to write 'delete' – have to mark variables/fields as owned/shared/borrowed
+ manageable implementation
+ low impact on execution-time program performance

Strawman Design

Outline for Strawman Design

New Class Value Kinds

- **4 different kinds of class values:**
  - 'owned', 'shared', 'unmanaged' and 'borrowed'
  - each is actually a different type
  - there used to be just 1 kind that was similar to 'unmanaged'
- **'new' call can specify which kind of value to create:**

```
class MyClass { ... }
var a: unmanaged MyClass = new unmanaged MyClass();
// 'a' refers to a manually managed instance that needs to be 'delete'd at some point
var b: owned MyClass = new owned MyClass();
// the instance referred to by 'b' is deleted at end of scope
var c: shared MyClass = new shared MyClass();
// the instance referred to by 'c' is reference counted
var d: borrowed MyClass = new borrowed MyClass();
// the instance referred to by 'd' will be deleted at the end of scope
```

New Class Value Kinds: Type Inference

- **What if the variable types are left out?**

- **Type inference works as one might expect:**

```
class MyClass { ... }
var a                         = new unmanaged MyClass();
// 'a' refers to a manually managed instance that needs to be 'delete'd at some point
var b                     = new owned MyClass();
// the instance referred to by 'b' is deleted at end of scope
var c                      = new shared MyClass();
// the instance referred to by 'c' is reference counted
var d                        = new borrowed MyClass();
// the instance referred to by 'd' will be deleted at the end of scope
```

More About Borrowed

- **A borrow is**
  - a pointer to a class instance...
    ... that does not impact the lifetime of the instance
- **Class types default to 'borrowed'**
  - 'MyClass' is the same as 'borrowed MyClass'
  - Expect that borrowed is appropriate for most uses of classes
- **Several ways to borrow from a managed class value:**

```
class MyClass { ... }
var x = new owned MyClass();
// The following are equivalent ways of declaring a borrow from x:
var b = x.borrow();
var b: MyClass = x.borrow();
var b = x: MyClass; // Cast to borrow
var b: MyClass = x; // Coerce to borrow
```

Coerions to Borrowed

- **Coercions from 'owned' to 'borrowed' keep code simpler:**

```
proc compute(input: MyClass) { ... }
// Could be written as
proc compute(input: borrowed MyClass) { ... }

var x = new owned MyClass();
compute(x); // Coerces to borrow to pass argument
```

- Similar coercions also available for ‘shared’ and ‘unmanaged’

new MyClass

- **What happens with an undecorated 'new'?**

```
class MyClass { ... }
var a = new MyClass();
```

- **Here the type of 'a' is a 'borrowed MyClass'**
  - the instance will be destroyed at the end of scope
  - returning 'a' results in a compilation error

- **This choice keeps type inference consistent:**

```
var a = new MyClass();
var a: MyClass = new MyClass();
```

- **The following are also equivalent to the above:**

```
var a: MyClass = new owned MyClass();     // coercing to borrow
var a = (new owned MyClass()): MyClass;  // casting to borrow
var a = (new owned MyClass()).borrow();
```

Class Subtyping

- **All class value kinds support subtyping**
  - This example shows 'owned', but 'shared' and 'unmanaged' work too

```
class ParentClass {
  proc parentMethod() { ... }
}
class ChildClass: ParentClass { ... }

proc consumeParent(arg: owned ParentClass) { ... }
var x = new owned ChildClass();
consumeParent(x); // coerces 'owned ChildClass' to 'owned ParentClass'
                  // and consumes x, leaving it 'nil'

proc borrowParent(arg: ParentClass) { ... }
var y = new owned ChildClass();
borrowParent(y); // coerces 'owned ChildClass' to 'borrowed ParentClass'
                 // y still stores an object
```

Class Methods

- **Class methods borrow 'this'**

```
proc MyClass.method() {
  writeln(this.type:string); // outputs the borrow type 'MyClass'
                             // a.k.a. 'borrowed MyClass'
}
```

- **Coercions to borrow enable method calls on 'owned'**

```
var x = new owned MyClass();
x.method(); // 'this' argument coerces to borrow in call
```

- **Future work: indicate 'this' is 'unmanaged' or 'owned' ?**

Borrowed Arguments Don't Impact Lifetime

- **An argument with borrowed type does not impact lifetime**
  - it would be an error to save the borrow into a global, e.g.
  - it would be an error to delete one
  - many of these errors are raised at compile-time
- **For example:**

```
var global: borrowed MyClass; // 'borrowed' optional here
proc saveit(arg: borrowed MyClass) { // and here
  global = arg; // Error! trying to store borrow from local 'x' into 'global'
  delete arg;   // Error! trying to delete a borrow
}
proc test() {
  var x = new owned MyClass();
  saveit(x); // x coerced to borrow on call
  // instance destroyed here
}
test();  writeln(global);   // uh-oh! use-after free
```

Owned/Shared Arguments Impact Lifetime

- **A default-intent 'owned' argument transfers ownership**
  - otherwise, removing it from the formal argument would be equivalent...
    ... and then why bother writing 'owned' at all?
  - Note, Owned and Shared previously required 'in' intent for this
- **For example:**

```
var global: owned MyClass;
proc saveit(arg: owned MyClass) {
  global = arg; // OK! Transfers ownership from 'arg' to 'global'
  // now instance will be deleted at end of program
}
proc test() {
  var x = new owned MyClass();
  saveit(x); // leaves x 'nil' - instance transferred to arg & then to global
  // instance not destroyed here since x is 'nil'
}
test();  writeln(global);  // OK
```

New Compile-time Checking

- **Lifetime checker is a new compiler component**
  - It checks that borrows do not outlive the relevant managed variable
- **For example, this will not compile:**

```
proc test() {
  var a: owned MyClass = new owned MyClass();
  // the instance referred to by a is deleted at end of scope
  var c: MyClass = a.borrow();
  // c "borrows" to the instance managed by a
  return c; // lifetime checker error! returning borrow from local variable
  // a is deleted here
}
```

```
$ chpl ex.chpl --lifetime-checking
ex.chpl:1: In function 'test':
ex.chpl:6: error: Scoped variable c cannot be returned
ex.chpl:2: note: consider scope of a
```

Generic Arguments

- This section describes elements of the design that are less solid

Generic Arguments Default to Borrowed

- **Totally generic arguments don't transfer ownership**
  - e.g. 'proc f(arg)' or 'proc f(arg: ?t)'
  - Ownership transfer for such functions would be surprising

```
proc f(x) { ... }
var x = new owned MyClass();
f(x);
writeln(x); // Surprising if this outputs 'nil'
```

  - Function signature should show potential for ownership transfer
    - so library users can understand APIs

- **Instead, such generic arguments need to opt in:**

```
proc f(x) { ... }
f(new owned MyClass()); // f gets a borrow

proc g(x: owned) { ... }
g(new owned MyClass()); // g takes ownership
```

Exceptions to the Rule

- Type arguments do not default to borrow:

```
proc printType(type t) {
  writeln(t: string);
}
printType(owned MyClass);
// outputs 'owned MyClass'
```

- Compiler-generated initializers do not default to borrow:

```
record Container {
  var field;
}
var y = new Container(new owned MyClass());
// y has type Container(owned MyClass)
```

A flexible generic argument?

- **How to write a generic function that**
  - accepts both 'owned' and 'borrowed' class values
  - …and leaves it up to the caller which type is used?
- **'managed?' keyword is the strawman proposal**
  - indicates to compiler that it should not transform argument to borrow

```
proc h(x: managed?) { ... }
h(new owned MyClass()); // 'h' takes ownership
h(new owned OtherClass()); // 'h' takes ownership
h(global.borrow()); // or 'h' can borrow
h(1); // 'h' can also apply to non-class things
```

Generic Collection Example

- **Consider a simplified generic collection:**

```
record Collection {

  var element: ...;
}
proc Collection.addElement(arg) {
  element = arg;
}
```

Generic Collection Example: owned only

- **What if the Collection wanted to accept *owned* only?**

```
record Collection {

  var element: owned;
}
proc Collection.addElement(arg: owned) {
  element = arg;
}
```

- **Now addElement does ownership transfer**

```
var c: Collection(owned MyClass);
c.addElement( new owned MyClass() ); // transferred to element
```

- **But the collection can't store an int or a borrow**

```
var d: Collection(int);      d.addElement( 1 ); // errors
var e: Collection(MyClass); e.addElement(global.borrow()); // errors
```

Generic Collection Example: borrow only

- What if the Collection wanted to accept *borrow* only?

```
record Collection {

  var element: borrowed;
}
proc Collection.addElement(arg: borrowed) {
  element = arg;
}
```

- Now addElement does not transfer ownership

```
var c: Collection(MyClass);
c.addElement( global.borrow() ); // collection borrows global
c.addElement( new owned MyClass() ); // collection borrows new owned class
```

- But the collection can't store an int:

```
var d: Collection(int);      d.addElement( 1 ); // errors
```

Generic Collection Example: unspecified

- **What if the collection uses a totally generic field?**

```
record Collection {

  var element;
}
proc Collection.addElement(arg: element.type) {
  element = arg;
}
```

- **Collection can store anything**
  - owned, shared, borrowed, record, int, ...
  - since *Generic Arguments Default to Borrowed* rule does not apply to:
    - type arguments (relevant to the compiler-generated type constructor)
    - compiler-generated initializers
    - arg:element.type (generic but has specified type)

Generic Collection Example: unspecified

- **Here is alternative way of writing the same:**

```
record Collection {
  type elementType;
  var element: elementType;
}
proc Collection.addElement(arg: elementType) {
  element = arg;
}
```

- **Collection can still store anything**

```
var c: Collection(owned MyClass);
c.addElement( new owned MyClass() ); // transferred to element

var d: Collection(int);     d.addElement( 1 ); // OK
var e: Collection(MyClass); e.addElement(global.borrow()); // OK
```

Generic Args Default to Borrow: Alternatives

- **Why include a *generic argument defaults to borrow* rule?**
  - avoid surprise in a case like this:

```
proc f(x) { ... }
var x = new owned MyClass();
f(x);
writeln(x); // Surprising if this outputs 'nil'
proc f(x) { writeln(x); }
```

  - 'managed?' is simply a way to opt-out of that behavior

- **Would compile-time checking for 'nil' 'owned' be better?**
  - Would need an owned that can store nil for use as array element
  - Such a nil-able owned would still be subject to above confusion

- **Should 'managed?' be more similar to intents ? 'owned' ?**

Progress towards Prototype

Progress towards Prototype

- **In version 1.17**
  - coercions from Owned(C)/Shared(C) to C
  - coercions from Owned(SubClass) to Owned(ParentClass)
  - array push_back() now works for Owned
  - lifetime checker is available but off by default
    - activate with --lifetime-checking
- **In branches**
  - explored 'new' returning 'owned' class values by default
    - leaning away from this due to type inference inconsistency
  - distinguish between 'borrowed' and 'unmanaged' class values
  - enable 'new owned C' / 'new unmanaged C'
- **Not started yet**
  - compile-time checking for 'nil' owned
  - 'nil'-able owned
  - strategy for raising an error when a borrowed object is invalidated

Questions Answered

- **Does lifetime checking make Chapel too hard to use?**
  - The answer appears to be "no" *for incomplete checking*
  - Ran lifetime checker on all tests on master branch
  - A relatively small amount of code needed updating
- **Will this effort require Chapel users to adjust programs?**
  - The answer is "yes" for programs using classes
  - The main issue is using unmanaged/owned/shared/borrowed
  - Plain class type ('MyClass') is changing meaning
    - from 'unmanaged' to 'borrowed'
  - Expecting all class 'new' calls must be decorated at first
  - Old behavior possible by using 'unmanaged' in many places
    - class types
    - new statements

Open Questions

Open Questions

- **Is this language design direction a good path?**
- **Is 'shared' by default better than 'borrowed' by default?**
- **How should 'owned', 'borrowed' apply to non-class types?**
  - e.g. can a record or integer pass to an 'owned' argument?
- **Finalize language design around generic instantiation**
  - 'managed?' is a straw-man keyword
  - open questions about type arguments, type constructors, & initializers
- **How to make 'this' in a method be 'unmanaged'?**

Open Questions

- **Should 'owned' be able to store 'nil'?**
- **What is the syntax for specifying lifetimes?**
  - the compiler can often infer lifetimes
  - explicit syntax is required for the cases where inference is insufficient
- **Should there be "safe" and "unsafe" blocks? functions?**
  - would it be an error to call an unsafe function from a safe block?
  - what would the syntax be?

Function Hijacking

Hijacking: Background

- **Function hijacking leads to surprising behavior:**
  - For example, suppose an application developer uses a library:

```
// Library
module Lib {
  var global: int;
  proc setup() {
    writeln("in Lib.setup()");
    global = 1;
  }
  proc run(x) {
    setup();
    writeln("Global is ", global);
  }
}
```

```
// Application
module App {
 use Lib;
 proc main() {
   run(1);
 }
}
```

- **Output:**

```
in Lib.setup()
Global is 1
```

OK!

Hijacking: Background

- **Now suppose the application adds a 'setup' function:**

```
// Library
module Lib {
  var global: int;
  proc setup() {
    writeln("in Lib.setup()");
    global = 1;
  }
  proc run(x) {
    setup();
    writeln("Global is ", global);
  }
}
```

```
// Application
module App {
 use Lib;
 proc main() {
   run(1);
 }
 proc setup() {
   writeln("in App.setup");
 }
}
```

- **Output:**

```
in App.setup
Global is 0
```

Uh-Oh! global not set!

Hijacking: This Effort, Impact, Next Steps

This Effort: Investigated function hijacking scenarios in Chapel

- Explored 7 scenarios
  - starting with the example we just saw
- Described why problems occur
- Planned language design directions to solve these problems
  - 'override' keyword for methods
  - pure virtual methods
  - constrained generics
- CHIP 20 contains the details!

Impact: Serves as a starting point for language design

Next steps: Improve language design and implementation

Open Fabrics Interface ('ofi') Communication Layer

‘ofi’ Comm Layer

Background: Want a portable high-performance comm layer

- Highest-performance ones (ugni) have not been portable
- Portable ones (GASNet) lacked performance on specialized networks

This Effort: Create a comm layer based on libfabric (OFI)

- Hope: network vendors will create high-performance providers
- Chapel can maintain fewer comm layers but retain performance
- Design work and and stubbed implementation complete

Impact: Performance portability at last?

- For this special case, anyway

Next Steps: Ongoing effort, aiming for delivery this year

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