Transitioning from Constructors to Initializers in Chapel

Lydia Duncan and Mike Noakes, Cray Inc.

CHIUW 2018

May 25, 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.

Constructors were less than ideal

- **Served their purpose but had some issues, including:**
  - Required an argument per generic field
    - Purely for compiler use, couldn't change value of param/type fields in body

```
proc MyType.MyType(type t, x) {
  this.x = x;
}
```

  - User constructors always called default constructor
    - Could result in double initialization of fields (or more)
  - Default constructor always generated
    - User couldn't prevent creating an instance by specifying all fields
  - Lack of clarity on when copies are made
  - Many other issues

Developed initializers as a replacement

- **Can avoid duplicate initialization**
  - User initializers do not call a default initializer
- **One name across all types**
  - Not as fragile w.r.t. type name changes
- **Arguments not tied to fields, so can omit any subset**

```
proc init(val) {  // t field no longer requires an argument
  t = val.type;   // type and param fields can be set in the body like other fields
  x = val;
}
```

- **Overall, just better!**

Initializer structure

```
// from modules/internal/ChapelError.chpl
class IllegalArgumentError : Error {
  var formal: string;
  var info: string;

  proc init(info: string) {
    // super.init();
    // this.formal = "";
    this.info = info;
  }
}
```

Field initialization occurs in order, starting with inherited fields

If no explicit call to the parent initializer, one with zero arguments will be inserted

Skipped fields on the current class are given their declared value if provided, or the default value for the type.

Inheritance example

```
// from modules/internal/DefaultRectangular.chpl
class DefaultRectangularDom: BaseRectangularDom {
  var dist: DefaultDist;
  var ranges : rank*range(idxType,bounded,stridable);

  proc init(param rank, type idxType, param stridable, dist) {
    super.init(rank, idxType, stridable);
    this.dist = dist;
    // ranges initialized here
  }
}
```

Inherited fields can be used in the type or default value of fields on the current class

They must be initialized via a parent initializer call prior to any fields in the child.

Again, if no explicit call is provided, one will be inserted if possible.

Calling other initializers defined on the type

```
// from modules/standard/DateTime.chpl
record timedelta {
  … // fields, etc.

  proc init(timestamp: real) {
    this.init(seconds = timestamp: int,
              microseconds=((timestamp - timestamp: int)*1000000): int);
  }

  proc init(days=0, seconds=0, microseconds=0,
            milliseconds=0, minutes=0, hours=0, weeks=0) { … }
}
```

For code reuse, calls the other initializer

This initializer performs the brunt of the work

Method calls

- **Methods can only be called on initialized things**
  - Until all fields on current type are initialized, could give bad behavior
  - Initializers forbid method calls prior to super.init() call
    - After super.init(), can call methods defined on parent types
  - Signal that the current type is initialized with this.complete()
    - Allows methods on current type, but not full dynamic dispatch

Method call / this.complete() example

```
// from modules/internal/String.chpl
record string {
  … // fields, other initializers, etc.

  proc init(cs: c_string, length: int = cs.length,
            isowned: bool = true, needToCopy:  bool = true) {
    this.isowned = isowned;
    this.complete();
    const cs_len = length;
    this.reinitString(cs:bufferType, cs_len, cs_len+1, needToCopy);
  }
}
```

Initializes all remaining fields

The method called helps set up the type, but is used in other places so it relies on the fields having an initial value.

Can also send `this` as argument to other functions once fields are ready

Summary of init()

- **Given a class hierarchy:**
  - Classes A through D form a hierarchy: D:C:B:A
- **In D.init(), object starts as nothing (a blob of memory)**
  - Implication: You can't do much with it yet
- **After D's call to super.init(), object is a C**
  - Implication: You can do anything with it that you could do with a C
  - Plus, you can also assign to D fields to help turn it into a D
- **Object becomes a D:**
  - After D's call to D.complete(), or
  - After D's call to this.init(), or
  - After D.init() returns

postinit example

```
// helper class from modules/dists/BlockCycDist.chpl
class LocBlockCyclicDom {
  param rank: int;
  type idxType;
  param stridable: bool;
  const globDom: BlockCyclicDom(rank, idxType, stridable);
  var myStarts: domain(rank, idxType, stridable=true);
  var myFlatInds: domain(1);
}

proc LocBlockCyclicDom.postinit() {
  myFlatInds = {0..#computeFlatInds()};
}
```

Type defines lots of fields

Only one needs a different value than the declared

Could redefine default initializer, but not really worth it, so use a postinit method instead

postinit example

```
// helper class from modules/dists/BlockCycDist.chpl
class LocBlockCyclicDom {
  …
}

proc LocBlockCyclicDom.postinit() {
  myFlatInds = {0..#computeFlatInds()};
}
```

postinit runs *after* any initializers, including ones on type that inherited from this one.

Because of this, it can call methods

Note: dynamic dispatch now takes effect

Copy initializers

- **Copies of record contents occur in several places:**

```
var x: R = …;
var y = x; // copies x

proc foo(in val: R) { … } // copies argument
```

- **Could change a type's copy behavior in constructor world**
  - But finicky and complex

Copy initializer example

```
// From modules/standard/IO.chpl
record file {
  var home: locale = here;
  var _file_internal:qio_file_ptr_t = QIO_FILE_PTR_NULL;
}

proc file.init(x: file) {
  this.home = x.home;
  this._file_internal = x._file_internal;
  this.complete();
  on home {
    qio_file_retain(_file_internal);
  }
}
```

Can write special kind of initializer called a copy initializer

The compiler provides a default copy initializer if no explicit one is provided, which just copies all fields

In this particular case, we want to also reference count the _file_internal field.

Initializers: Status

- **Initializers are overall in pretty good shape**
- **Most library/internal modules converted to initializers**
  - Exception:
    - Shared: special initCopy/autoCopy functions not converted to copy init
- **Most tests converted to initializers**
  - Out of ~8,500 tests...
    ... 26 remain unconverted due to bugs or unimplemented features
    ... 28 others will be removed once constructors are deprecated

Initializers: Next Steps

- **Finish compiler-generated initializers**
  - Fix internal compiler errors
  - Some expressions cannot be used as default values for fields yet
    - E.g., parallel loops, conditional expressions
- **Fix bugs**
  - Nested types when at least one of the types is generic
  - Generic instantiation when generic fields initialized in conditional
  - …
- **Deprecate constructors**

Initializers: Next Steps

- **Finalize design decisions:**
  - Finalize copy initializers
  - Finalize type initializer story
    - User might want to permit shorter type declarations, for instance:

```
var twoTypes: SomeType(int, real);
var bothSameType: SomeType(int);
```

  - Allow users to opt into retaining compiler's default 'init()'?
    - Currently squashed by user's 'init()'
- **Support incomplete initialization when explicitly requested**
  - Also known as the 'noinit' feature

Other contributors

- **Special thanks to:**
  - Kyle Brady
  - Brad Chamberlain
  - Michael Ferguson
  - Ben Harshbarger
  - Tom Hildebrandt
  - Vass Litvinov
  - And the rest of the Chapel Team, past and present!

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

THE SUPERCOMPUTER COMPANY