1 Classes

[Sections unrelated to constructors have been omitted.]

1.1 Constructors

Constructors are special methods which perform object initialization. They are provided by Chapel to support object-oriented programming. The properties which make them special are:

- A constructor is always invoked automatically during object creation; and
- The compiler automatically provides default initialization in the definition of each constructor.

Taken together, these properties ensure that all of the fields in a new object are initialized to a predictable value before the object is available for use.

The remainder of this section is organized as follows: The succeeding three subsections discuss constructor declarations (§1.1.1), constructor invocations (§1.1.2) and constructor semantics (§1.1.3). The final subsection details the differences between this new definition of constructors and the existing implementation.

1.1.1 Constructor Declarations

A class in Chapel always has at least one constructor associated with it: If no user-defined constructor is supplied, the compiler generates one automatically. The compiler-generated constructor is described in section §1.1.3.1.

A user-defined constructor is a constructor method explicitly declared in the program. Such a constructor declaration has the same syntax as a method declaration, except that it is introduced using the `constructor` keyword and there is no return type specifier. Constructor declarations do not have a parentheses-less form.

User-defined constructor declarations have the following syntax:

constructor−declaration−statement:
  linkage−specifier$_{opt}$ **constructor** *type−binding constructor−name argument−list*
    where−clause$_{opt}$ *constructor−body*

constructor−name:
  identifier

constructor−body:
  statement

Aside from the `constructor` keyword, the syntax elements of a constructor declaration — including *linkage−specifier*, *type−binding* and *where−clause* — have the same syntax as for a method declaration. Statements within a *constructor−body* have the same syntax as a *function−body*, except that the `init` statement is permitted only within a constructor body.

> *Implementors' note*. It is probably simplest to allow the `init` statement everywhere syntactically, and then emit a diagnostic message if it appears outside the context of a constructor body.

It is illegal to declare a return type for a constructor; the return type is implicitly `void`. This also means that if a `return` statement appears within the *constructor−body*, it must be the form that does not return an expression.

1.1.2 Constructor Invocation

Constructors may be invoked in two ways: as part of object creation, or as a method. An object is created using a *new−expression*:

new−expression:
  new *constructor−call*

constructor−call:
  class−name . *constructor−name argument−list*
  class−name argument−list

class−name:
  identifier

The *constructor−call* resembles and behaves like a function invocation. If the *constructor−name* is omitted, the default constructor name `__init__` is used. Constructor invocations do not have a parentheses-less form. Normal function name lookup and argument resolution (§**??**) are applied as if the *constructor−call* is a static function invocation. However, the selected function must be a constructor; otherwise, a compile-time error is issued.

The execution of a `new` expression allocates memory sufficient to hold the object being constructed. This uninitialized object is passed as the receiver argument to the contructor named in the `new` expression. The matching constructor performs its initialization operations on the object (as a method invocation). The return value of the `new` expression is that same object after it has been modified by the constructor.

A constructor may also be invoked like a normal method. When invoked as a method, a constructor simply re-initializes the receiver object.

> *Example (newEquivalence.chpl)*. Given the class declaration,
>
> ```
> class C {
>   var _bar : int;
>   constructor __init__(bar: int) { _bar = bar; init; }
> }
> ```
>
> using `new` to create an initialize an object

```
var c = new C(3);
```

is equivalent to

```
extern proc malloc(size:int):opaque;
var c = malloc(__primitive("sizeof", C)) : C ; // Some memory cast to type C.
c.__init__(3);
```

In the context of a derived class constructor, the interpretation of the base class constructor call as a method invocation is consistent, since the base-class sub-object is not yet initialized, and it is referred to through the `super` field name.

Example (baseCconstructorCall.chpl).

```
class B {
  var _a,_b: int;
}
class D : B {
  var _c, _d: real;
  constructor(c:real, d:real) {
    _c = c; _d = d;
    super.__init__(_c:int, _d:int); // _a <- _c:int; _b <- _d:int;
  }
}
```

Whether or not the `new` operator is used, the default constructor name `__init__` may be omitted. Thus the above base-class constructor call can also be expressed as `super(_c:int, _d:int);`.

> *Open issue*. Assuming that a constructor can be called like a normal method, zero-initialization of the memory allocated by the `new` operator becomes desirable. As long as valid field values differ from their zero-initialized values, it then becomes possible for the constructor to detect whether it is being called to perform initialization or to perform reinitialization. The distinction is important where memory tracking might be performed. In the reinitialization case, it would be important to release one object (reference) before replacing it with another.

Constructors for generic classes (§**??**) have `param` and `type` arguments. Such arguments and their corresponding fields are handled similarly to arguments and fields that contain runtime values. However, assignments to a `param` or `type` fields may only appear within the initialization clause of a constructor. They may not appear in the body of a constructor or normal method. See Section **??** for additional details.

1.1.3 Constructor Semantics

The construction of an object proceeds in two phases: the initialization phase and the construction phase. The initialization phase establishes an initial value for each field in the object (including the base-class sub-object). The construction phase performs whatever other actions are necessary to establish the class invariants. These two phases are separated by the `init` statement. If no `init` statement appears within a constructor, the compiler inserts one automatically just before the end of the constructor body.

In the initialization phase, some special rules apply:

- Method calls involving `this` (i.e. the object being constructed) are not permitted.
- Statements which appear to be assignments but which refer to a field on the left-hand side are treated as initializations rather than assignments.[1]
- References to field names access the raw data stored in that field, they do not invoke field accessor functions.
- The base-class sub-object may be initialized and referred to, but has the run-time type of that base class.
- A base-class sub-object must be initialized as a whole before any of its fields may be referenced.

In the construction phase, any statement which can normally appear in the body of a method of that class is permitted.[2]

When the `init` statement is encountered, the compiler supplies default initializers (§1.1.3.2) for any fields which have not yet been initialized explicitly. Since Chapel is a compiled language, it is possible for the compiler to determine through static analysis whether an `init` statement has been encountered along the current branch of execution before an attempt to call a method on `this`. That permits such an error to be diagnosed at compile time. It follows naturally that the compiler can determine statically which fields have been initialized and which have not along any execution path. Execution paths may reconverge, and may have different sets of initialized fields associated with them. This creates an ambiguity which is reported as an error at compile time.

Within a constructor, the `this` keyword may be used to disambiguate field names from argument names. However, prior to an `init` statement, `this` is used only to establish the scope of the (field name) identifier which follows; it is not interpreted as a reference to the object being constructed.

It should be noted that special initialization actions required by certain types (such as sync and array variables) can be handled easily within a constructor. In particular, it is permissible to call methods on a given field once it has been initialized; only calls to methods on the object as a whole (`this`) are prohibited, and then only prior to encountering an `init` statement. Thus (for example), the code required to initialize a sync variable can be called as part of its explicit initialization.

Similarly to fields having a run-time value associated with them, `param` and `type` fields may be initialized within an initialization clause.

Example (fieldInitializers.chpl). This example shows a simple class declaration containing a user-defined constructor.

```
class Point {
  var x,y : real;
  constructor Point(a : real, b: real)
  { x = a;          // Initializes field x with the value a.
    this.y = b;     // Initializes field y with the value b.
    /* init; */     // The compiler inserts an init statment here, if needed.
  }
}
```

[1] If "reference assignment" is added to the language, it will be unnecessary to handle field assignments specially in this context.

[2] In practice, there may be several stages within the construction phase — it being valid to call only certain subsets of the methods within a class at each stage. Keeping track of which invariants are required for and established by each of these methods is left entirely to the user.

Because they do not reference the object being created, static methods may be invoked anywhere within a constructor body.

Rationale. We introduced the `init` statement to mark the initialization point and perform initialization. The portion of the *constructor–body* following an `init` statement can perform common post-initialization steps. (Those steps *could* be performed at the call site if the `init` statement were not supplied. However, good coding practice suggests migrating these common steps into the constructor itself.)

The compiler must already perform static liveness analysis to determine if all fields have been initialized. Once this is done, it can supply missing initializations automatically. Furthermore, it requires no more work to perform this test at arbitrary positions within the constructor body — at the `init` statement — than at the end.

Open issue. The naming convention for constructors has not been determined. If we follow the C++ model, then we do not need a new keyword. We can use the `proc` keyword to introduce a constructor, and then key off the fact that the name of the procedure matches the name of the class.

The alternative — using a `constructor` (or equivalent) keyword — permits the constructors in a class to be named arbitrarily. This would have the disadvantage of making constructor calls more verbose in general, since they would need to be qualified by the class name. For example:

```
var myFoo = new foo.create();
```

On the other hand, it enables constructor names which are much more descriptive. For example:

```
constructor foo.copy(oldFoo: foo)
{ a = oldFoo.a; b = oldFoo.b; }
```

This represents a copy constructor (in the C++ sense). One could use similarly mnemonic names for default constructors, cloning constructors, pointer-stealing constructors and so on. A remedy for the verboseness of this latter approach is to select a stereotyped name (for example `__init__`) which is implied if just the class name is supplied in a constructor call. Thus `var myFoo = new foo();` would be equivalent to `var myFoo = new foo.__init__()`.

Example (constructor.chpl). The following example shows a class with two constructors:

```
class MessagePoint {
  var x, y: real;
  var message: string = "I'm a point.";

  constructor byLocation(x: real, y: real)
  { this.x = x; this.y = y; }

  constructor byMessage(message: string)
  { this.message = message; }
}  // class MessagePoint

// create two objects
var mp1 = new MessagePoint.byLocation(1.0,2.0);
var mp2 = new MessagePoint.byMessage("point mp2");
```

The first constructor lets the user specify the initial coordinates and sets the string to the default value specified in the class declaration: `"I'm a point."`. The second constructor lets the user specify the initial message when creating a `MessagePoint`, and sets both coordinates to their type-dependent default value: `0.0`.

Whether called implicitly or explicitly, a base-class constructor runs to completion before returning. This is consistent with treating the base-class sub-object as if it were a separate field. It is also consistent with the behavior the syntax suggests: Since a constructor can be called like a normal method, it is natural to assume that its behavior is the same when called from another constructor.

1.1.3.1 The Compiler-Generated Constructor

If no user-defined constructor is supplied for a given class, the compiler provides one. The compiler-generated constructor uses the default constructor name, so it is invoked when just the class name is supplied in a *new−expression*.

The compiler-generated constructor is defined as follows. Its argument list contains one argument for each field defined in the class. Each argument is named identically with the corresponding field, and has a default value which is provided by the field initializer in the class declaration if present, and otherwise by the type-dependent default value.§1.1.3.2 The arguments appear in the order in which the correponding fields appear in the class declaration (i.e. in declaration order).

The body of the compiler-generated constructor initializes each field with the value of the corresponding argument. These initializations appear in the order in which the fields were declared in the class declaration (i.e. in declaration order). Note that default initialization is not required, because every field is initialized explicitly.

If the class is a derived class, the following additions are made: The argument list is expanded by prepending arguments corresponding to the fields in the base class and its base class, and so on recursively. This is done so that the fields belonging to the most ancient ancestor class appear first and in the order declared in that class. The fields from the next-most ancient ancestor then appear — also in declaration order — and so on. Any ancestor field name which is shadowed by a field of the same name in one of its descendent classes is omitted from the argument list. The body contains a base-class constructor call as its first statement. All of the formal arguments not used to initialize fields in the most-derived class are passed as named arguments to the base-class constructor — in the same order in which they appear in the compiler-generated constructor. Note that even though shadowed field initializers are not passed in explicitly, guaranteed initialization is still achieved because argument defaults are supplied for those missing arguments.

Example (compilerGeneratedConstructor.chpl). Given the class

```
class C {
  var x: int;
  var y: real = 3.14;
  var z: string = "Hello, World!";
}
```

the compiler will generate a constructor equivalent to the following:

```
constructor C.__init__(x:int = 0, y:real = 3.14, z:string = "Hello, World!")
{ this.x = x; this.y = y; this.z = z; init; }
```

The `x` argument has the type-dependent default value of `0`, since no initializer for field `x` is supplied in the class declaration. The `y` and `z` arguments have the default values `3.14` and `"Hello, World!"`, copied from the initializer expressions in their respective field declarations.

Because argument defaults are provided as part of the compiler-generated constructor, some, none or all of the default field initializers can be overridden in a constructor invocation.

Example (callingGeneratedConstructor.chpl). For example, instances of the class `C` defined above can be created by calling the compiler-generated constructor as follows:

- The call `new C()` is equivalent to `C(0,3.14,"Hello, World!")`.
- The call `new C(2)` is equivalent to `C(2,3.14,"Hello, World!")`.
- The call `new C(z="")` is equivalent to `C(0,3.14,"")`.
- The call `new C(2, z="")` is equivalent to `C(2,3.14,"")`.
- The call `new C(0,0.0,"")` specifies the initial values for all fields explicitly.

Example. As an additional example, let us consider a derived class.

```
class B {
  var a,b,c,d,e,f: int;
}
class D : B {
  var d,e,f,g,h: real;
}
```

The compiler-generated constructor for class `D` will be equivalent to:

```
constructor D.__init__(a:int = 0, b:int = 0, c:int = 0,
                       d:real = 0.0, e:real = 0.0, f:real = 0.0,
                       g:real = 0.0, h:real = 0.0)
{ super.__init__(a = a, b = b, c = c);
  this.d = d; this.e = e; this.f = f; this.g = g; this.h = h; }
```

Note that this implies the existence of a default-named base-class constructor taking three arguments. The compiler-generated constructor for class `B` will fulfill these requirements. Otherwise (i.e. if there are other user-defined constructors in the base class), the user must supply a matching constructor; otherwise, a resolution error will occur.

1.1.3.2 Default Initializers

Within a constructor, the compiler provides a default initializer for each field which has not been explicitly initialized. These initializers are inserted just in front of each `init` statement. If no `init` statement appears in the constructor body, the compiler automatically inserts one just inside the closing brace. If the constructor body is empty or contains a single statement, the `init` statment is inserted after the empty or simple body.

Default initializers are inserted for each field in declaration order (skipping those fields which have been initialized explicitly). If the class is a derived class and the base class sub-object has not been initialized, it is default-initialized by calling the default constructor on the base-class sub-object with zero arguments. Default initialization of the base-class sub-object precedes the default initialization of any field within the most-derived class.

Default initialization for a field will perform one of two actions depending on how the field is declared. If an initialization expression is supplied, that expression is evaluated and the result copied into the corresponding field. Otherwise, the field is initialized using a type-dependent default value.§**??**

1.1.3.3 The Default Constructors

As in C++, the default constructor for a class is a constructor which has the default constructor name and whose argument list is zero-length. Note that the compiler-generated constructor can always be called as a default constructor, since it provides a default value for each of its arguments.

1.1.4 Changes from Existing Behavior

This section enumerates the differences in syntax and semantics which this proposal represents, compared with the current implementation.

1.1.4.1 Syntax

At a high level, the only syntactical changes are the addition of the `constructor` keyword and the `init` statement. The restriction against calling methods on `this` before it is fully constructed depends on type information, so cannot be checked at parse time.

The effect upon the language and its use is discussed in more detail in the semantics section. But at a high level, the addition of the `init` statement provides a user with some new capabilities which were not available before:

- It provides an additional place to describe initialization semantics. The existing places were either as a constructor argument or in the field-initializer expression. Initialization semantics are those which occur before the object is fully initialized.
- It allows constructor arguments to be renamed before being used to initialize fields. Previously, initialization through constructor arguments was done using named arguments whose names match those of fields internal to the class (or record). Requiring the user to have knowledge of the (potentially private) field names breaks encapsulation.
- It provides a place where a base-class constructor call can be made such that its effects take place before field initialization of the derived class begins.

All of these effects expand what can be expressed by the language without taking away any existing capabilities.

1.1.4.2 Semantics

Quite a few semantical changes are implied by this proposal. These can be considered in terms of how they affect the definition of compiler-generated and user-defined constructors, how these are invoked, and how the addition of inheritance affects this description. In the following sections, we consider carefully the interaction between domains and arrays, and indicate any special handling required.

1.1.4.3 Allocation

Allocation is unchanged from the current implementation. The memory required to represent class objects is allocated from the heap, while that required to construct record objects is allocated on the stack.

1.1.4.4 Default Initialization

Default initialization is also unchanged from the current implementation, until the effect of the initialization phase is considered. In the current implementation, the point of initialization occurs in the default constructor wrapper, just before the constructor itself is called.[3] In the proposed implementation, it occurs just after the `init` statement.

Given that the compiler-generated constructor just initializes each field with its corresponding argument in declaration order, the body of a default constructor is redundant with the default wrapper, regardless how it is called. However, since a user-defined constructor can have completely different arguments and need not use its arguments at all, a user-defined constructor must be called after default wrapping supplies the requisite (zero and/or default) initialization.

In the new scheme, if the initialization phase is empty, the effect will be the same as for the current implementation, except that the construction phase of the compiler-generated constructor can now be truly empty. The part of the new specification stating that a default-initializer will be created for each field not explicitly initialized in the initialization phase essentially performs the same function as the default wrapper in the current implementation.

However, the addition of the initialization phase means that a constructor can also rename arguments before using them to initialize fields. It can also perform significant computation as part of that field initialization — computation that should perhaps not be exposed in a well-encapsulated class, or which would at least hinder the readability of client code.

> *Example (Initializer Clauses).* Let us compare constructors that could be used to convert between polar and rectangular coordinates: both avoiding initializer clauses (current implementation) and using them (proposed implementation).
>
> First we present some code that is common between the two implementations.

```
// This code is common between examples.

class Point { // Abstract base class
  proc x() { assert(false); }
  proc y() { assert(false); }
  proc r() { assert(false); }
  proc theta() { assert(false); }
}

class RectPoint : Point{
  var _x, _y: real;

  proc x() return _x;
  proc y() return _y;
  proc r() return sqrt(_x**2 + _y**2);
  proc theta() {
    param pi = 4.0 * arctan(1.0);
    var result: real;
    // Avoid numerical instability.
    if (abs(_y) < abs(_x)) then result = arctan(_y / _x);
    else result = pi / 2.0 - arctan(_x / _y);
    if ((_x < 0) && (abs(_y) < abs(_x)) ||
       ((_y < 0) && (abs(_x) < abs(_y)) then result += pi;
    return result;
```

[3] Due to the presence of the "meme" argument, a default wrapper is always created/called when the compiler-generated constructor is called, even when every argument is supplied explicitly.

```
  }
}

class PolarPoint {
  var _r, _theta: real;

  proc x() return _r * cos(_theta);
  proc y() return _r * sin(_theta);
  proc r() return _r;
  proc theta() return _theta;
}
```

In the current implementation, we then have:

```
// Polar <=> rectangular conversions, old style,
// using the compiler-generated constructor.

// Create a "native" rectangular point
var p: Point = new RectPoint(3.0, 4.0);

// Convert to a polar point
p = new PolarPoint(p.r(), p.theta());

// Convert back to rectangular.
p = new RectPoint(p.x(), p.y());
```

In each case, the default constructor is called, but the conversion is provided in the external code. The alternative is to construct points which initially represent the origin, and then flesh them out as part of the constructor.

```
// Polar <=> rectangular conversions, old style,
// using user-defined constructors.

proc RectPoint.RectPoint(p:Point)
  // (_x, _y) == (0.0, 0.0) here.
{
  // Danger! You can call methods on 'this' here,
  // but it's not ready yet.
  _x = p.x(); _y = p.y();
  // Fully-constructed here.
  /* Other stuff */
}

proc PolarPoint.PolarPoint(p:Point)
  // (_r, _theta) == (0.0, 0.0) here.
{
  // Same problem.
  _r = p.r(); _theta = p.theta();
  // Fully-constructed here.
  /* Other stuff. */
}
```

Note that with this approach, the class author can determine a point at which it is OK to call methods on the `this` being constructed. However, things get ugly in a hurry when inheritance is added. In that case, the field initialization for the base class must be factored out, and called directly (and explicitly) in the body of the derived class constructor. The base class sub-object is not placed in a consistent state until something is done explicitly in the derived class.

In contrast, the semantics in the proposed implementation are well-defined. The compiler itself enforces when it is OK to refer to each field and also when it is OK to refer to `this` as a whole. The point of initialization is clearly marked by the location of the `init` statement(s).

```
// Polar <=> rectangular conversions, new style.
```

```
constructor RectPoint.RectPoint(p:Point) {
   _x = p.x(); _y = p.y();
  init;  // Fully-constructed here.
  /* Other stuff */
}
constructor PolarPoint.PolarPoint(p:Point) {
  _r = p.r(); _theta = p.theta();
  init;  // Fully-constructed here.
  /* Other stuff */
}
```

In this case, each constructor puts the object in a consistent state by the time the `init` statement is reached. The difference in notation is slight, but the difference in the underlying computational model is significant.

The second important semantic difference pertains to user-defined constructors. In the current implementation, a user-defined constructor is reworked so that it calls the compiler-generated constructor with just its generic arguments. This ensures that default initialization is done on each field before the body of the user-defined constructor is entered.

However, that means that "meaningful" field initialization must be performed either in the field initializer expression or as an argument in the constructor call. The ability to perform a specific field initialization in a given overloaded version of the constructor is absent. Since the field initializer expressions provided in the class declaration are run in every constructor, there is no room for flexibility. The only way to obtain different behavior between one constructor call and another is through the passed-in arguments. As has been stressed above, this idiom breaks encapsulation.

In contrast, constructors in the proposed model have full control over how the fields are initialized. Each field can be initialized explicitly during the initialization phase. If so, the semantics of the constructor are shown explicitly, and are therefore easy to understand.

To provide "guaranteed initialization", the compiler supplies initializers for fields which are not initialized explicitly. It is important to note, however, that the compiler avoids double-initialization. The results are more intuitive, especially if the initialization expression in the class declaration has visible side-effects.

1.1.4.5 Construction

Construction is basically the same in the existing and proposed schemes. The existing implementation assumes that the object is fully-constructed by the time the constructor body is entered. The proposed implementation assumes that the object is fully constructed after the `init` statement is reached. The difference becomes apparent in considering the above example.

In the existing implementation, an attempt to provide distinct initialization behavior in different overloaded constructors will necessitate placing field assignments in the body of the constructor. That means that the point of initialization will occur somewhere *after* the start of the constructor body. Since the point of initialization is associated with an important assumption (namely, that in the suffix it is OK to call methods on `this`), it should be marked clearly with a comment or maintenance problems will ensue. The proposed `init` statement provides this indication. In addition, delaying default initialization to that point avoids the present problem of double-initialization.[4]

[4] In the existing implementation, default initialization is always called before the body of a user-defined constructor is entered. The default value can be overridden, but it is still initialized twice.

It is also important to note that assignment within the body of a constructor currently has *assignment semantics* as opposed to *initialization semantics*. The former has side effects as defined by the assignment operator for the type on the LHS.[5] The latter is a simple copy of the value of the RHS expression into the field named by the LHS. Since only assignment semantics are available in the body of a constructor in the current implementation, the expression of different kinds of initialization in different overloaded constructors really depends on the absence of side-affects associated withe the types of those fields. And therefore, the expressiveness of constructors and classes are somewhat limited.

In contrast, the present proposal makes the point of initialization easy to locate syntactically. It also provides a way to initialize the fields in an object explicitly, without triggering assignment semantics. We note, however, that assignment semantics can be invoked explicitly within the initialization clause (provided that interface is supported separately by the type associated with that field). Alternatively, the use of assignment can be postponed until the constructor body is executed.

1.1.4.6 Inheritance

Base-class constructor calls are supported in the proposed implementation. As with fields, since every constructor is assumed to provide guaranteed initialization, a call to a base-class constructor provides guaranteed initialization of the base-class sub-object. The compiler provides a base-class constructor call if one is not specified. Therefore, a derived class constructor need only concern itself with initializing the fields declared in the most-derived class.

In the existing implementation, constructors are supplied with an extra `meme` argument that carries the object being initialized. The body of a user-defined constructor contains just the actions specified in the constructor body. Default initialization is inserted by the compiler before the constructor is entered. In particular, the default constructor (the compiler-generated constructor with zero arguments) is called for each base class, starting with the most ancient ancestor. This guarantees that the object is default-initialized before the constructor body is entered.

Specification of a user-defined constructor in the base class causes the compiler-generated constructor to be hidden. Since every constructor in the base class calls the base-class compiler-generated constructor with a zero-length argument list, the compilation fails at this point: The compiler-generated base-class constructor cannot be found.

The compiler-generated constructor in the derived class contains an argument for each field in the base class (and its base classes, recursively). These are passed to the base-class constructor verbatim. So at least the default constructor implementation is self-consistent in the presence of inheritance.

1.1.4.7 Sync Variables, Domains and Arrays

Sync variables, domains and arrays have special assignment semantics defined in the internal library code whose invocation is arranged specially by the compiler. Special behavior is arranged through pragmas (which add flags to classes and funtions), and also through the class/record structure used to represent types which are known to the compiler.

[5] Some important examples of side effects are the change in the full-empty state of a sync variable, and the fact that assignment to an array results in an element-by-element copy.

Sync Variables carry the pragmas "sync", "no default functions" and "no object", corresponding to `FLAG_SYNC`, `FLAG_NO_DEFAULT_FUNCTIONS` and `FLAG_NO_OBJECT`. The lattermost merely prevents the `_syncvar` class from inheriting from `object`. It assumes that sync variables will always be statically typed and their methods statically dispatched. Therefore, no class ID is needed. The second flag means that sync variables do not receive the normal compiler-generated functions for reading and writing, equality testing, assignment, casting, copying and hashing. These are probably (or at least, should be) mere optimizations.

The first flag causes sync read semantics (i.e. `readFE`) to be applied when a sync argument is being coerced to a compatible (but different, i.e. non-sync) type. The flag also allows sync types to instantiate generics expecting a compatible non-sync type. In `insertFormalTemps()`, `chpl__autoCopy` is not called if the formal is a sync type. In insertReturnTemps(), the sync flag attached to the return value type causes read semantics to be inserted prior to the return.

The proposed constructor implementation should not affect these behaviors. It is observed that special semantics are generally inserted where there is a transition between a sync type and a non-sync type. Thus, care must be taken to propagate sync types faithfully to the argument types of any helper functions which are auto-generated by the implementation.

A recommendation for greater maintainability would depend on class-specific cast-in and cast-out functions. The compiler could then insert a cast where there was a transition between sync and non-sync types. By virtue of the specific casts implemented, the class could control the set of legal sync to non-sync coercions. In this way, all of the special behavior for sync variables that is currently wired into the compiler could be moved out into library code.

Domains, Arrays and Distributions are handled specially in several ways: They are reference-counted types, they are record-wrapped types and (except for distributions) they have runtime types.

Reference counting is an optimization added to allow reusing objects of those types where appropriate. The function `chpl__autoCopy` is redefined to increment a reference count contained in the abstract base class for the type. Assignment and copy operations are redefined to call `chpl__autoCopy()` and thus increment the reference count, and the destructor arranges to decrement the count.

The specialized code in the compiler relating to `chpl__autoCopy()` can probably be removed, provided that constructors, user-defined assignment operators and destructors provide hooks for maintaining the reference count, and provided that calls to these functions (particularly the destructor) are made in a consistent manner. As it stands, calls to `chpl__autoCopy()` are inserted by the compiler when creating a domain, when creating an array alias, when copying the contents of a tuple and when inserting a formal temp with blank intent. (These are in addition to the calls to `chpl__autoCopy()` made explicitly within the module code.)

The auto-copy function is excluded during the insertion of formal temps and ref temps — probably to avoid infinite recursion. It is known to preFold() so it can be folded out when its argument is a literal. It is used to control the insertion of explicit destroy calls — also an optimization. A better implementation would call the destructor invariably, and rely on inlining to remove calls to destructors which have trivial bodies. Some further specialized code relates to inlining this function in lowerIterators.cpp ("reconstruct autoCopy and autoDestroy for iterator records").

Record-wrapped types are implemented as classes, so that in general a field of that type can be replaced (to obtain polymorphic behavior). They are wrapped in a record mostly to force value semantics when they appear in an assignment (and possibly also in constructors, temp copies, argument and return value temporaries).

Arguments similar to those above (w.r.t. reference counting) can also be applied to record wrapping. Assuming the necessary hooks (as above) are provided for overriding the default [assignment and copy construction/initialization] behavior, and in addition we have a way to replace a reference to one instance of this type with another, then the semantics provided by the specialized code within the compiler can be pushed down into the module code.

The runtime type flag is applied to domains and arrays only (not distributions). It appears to be a way to provide type-dependent (a.k.a. polymorphic) behavior without relying on dynamic dispatch. Types passed around in runtime type wrapper records can be treated abstractly by the client code. However, it is still the case that type and its methods are resolved statically (during function resolution).

It may be that the current implementation of function resolution is not powerful enough to (automatically) provide the dynamic-to-static dispatch optimization, and this is a way to force that optimization. My understanding of this mechanism is not complete.

Provided that my analysis is correct, it should be possible to remove specialized compiler support for reference-counted types and record-wrapped types. All of the desired behavior provided by those two mechanisms — reference counting and by-value assignment semantics respectively — should be implementable entirely in module code. This hinges on the compiler actually generating calls to the appropriate copy constructor (or other initializer) for initialization, the assignment operator for assignment and the destructor when a local variable goes out of scope. Implementation of the classes which currently depend on the specialized code would have to be updated by overriding those three functions appropriately. The C++ spec and code which depends upon this overridable behavior can be used as a reference implementation.

Removal of support for "runtime types" from the compiler would depend on a full implementation of dynamic dispatch (if it is not already there), plus an additional optimization step to demote dynamic dispatches to static dispatches, where the target (receiver) type can be determined at compile time.

Example code is supplied below to demonstrate that the proposed constructor story is as powerful as the current implementation. The above excursion into specialized behavior attached to the domain, array and distribution types is mostly to assure us that nothing special needs to be implemented as part of the constructor itself — or alternatively to specify what accommodations must be made to duplicate existing behavior.

The simplest approach is to assume that dynamic dispatch works correctly, and that the compiler arranges to call copy-construction (or equivalent initialization code), assignment operators and destructors in a consistent manner — whereby the module programmer can override default behavior to provide the desire reference-counting and assignment semantics.

On the other hand, we can also argue that the proposed constructor semantics are at least as powerful as the existing ones. Implementation may be simpler if we can first demonstrate standard object-oriented behavior — having eliminated the specialized support code from the compiler. However, it is also possible to consider a "minimal change" approach: having the implementation support the existing specializations while expanding the constructor implementation to support the added syntax and semantics.

It all boils down to whether we can represent existing important code using the new syntax, and argue that the accompanying semantics will duplicate the present behavior. Hence, the following example.

> *Example (Array Creation).* It should suffice to show that the code exercised in creating an array will behave similarly under the present proposal. Given the declarations

```
config const n = 10;
var D : domain(1) = [1..n];
var A : [D] int;
```

the following code [allowing some poetic license] is created:

```
var tmp1:RTTI = chpl__buildDomainRuntimeType(defaultDist);
var tmp2:DefaultDist = tmp.d;
var tmp3:domain(1,int,false) = chpl__convertRuntimeTypeToValue(tmp2);
var tmp4:range(int,bounded,false) = _build_range(1,n);
var tmp5:domain(1,int,false) = chpl__buildDomainExpr(tmp4);
var D:domain(1,int,false) = tmp3 := tmp5;

var tmp6:domain(1,int,false) = chpl__buildDomainExpr(D);
var tmp7:RTTI = chpl__buildArrayRuntimeType(tmp6);
var tmp8:domain(1,int,false) = tmp7.dom;
var A:[domain(1,int,false)] int = chpl__convertRuntimeTypeToValue(tmp8);
```

We note that the declarations for `tmp1` through `tmp3` may currently be necessary to establish the static type of D. However, assuming that `chpl__buildDomainExpr()` can return a polymorphic object, the whole sequence can be reduced to

```
var tmp4:range = _build_range(1,n);
var D = chpl__buildDomainExpr(tmp4);
```

It is curious to note that the version of `chpl__convertRuntimeTypeToValue()` that creates the array A takes as its argument only the `.dom` portion of the runtime type information for the array. Yet, the returned array object has the right element type. It turns out that the generic form of that function is resolved with a type argument to a version that produces the correct array type. This mechanism is apparently somewhat hidden, but is consistent with the above description of "runtime types" as providing hooks for creating polymorphic behavior while supporting static type resolution.

It is suggestive, though not conclusive, that the mechanism of for creating a domain through `chpl__buildDomainExpr()` appears to require runtime type information only to establish the static type of the result (although in fact it also controls how that function is instantiated). The mechanism for creating an array from the given domain goes through `domain.buildArray()` and `domain.dsiBuildArray()`. The particular instantiation of `chpl__convertRuntimeTypeToValue()` controls the exact type of the array being created (as suspected). On the other hand, a polymorphic version of `buildArray()` or some similar function which was also generic w.r.t. element type could perform the same function at the top level (i.e. without depending on `chpl__convertRuntimeTypeToValue()`).

The conversion of the present runtime type mechanism to work with proposed constructors appears to be a no-op: the two seem to be orthogonal. The question of when fields are initialized and when the type building routines are called is taken care of by the parser and support code.

The remaining behavior which places a backward compatibility constraint on the new constructor proposal is the order of initialization of the components of a class which contains and uses a domain description. The reason why default initialization to zero does not work in these cases is that an array cannot be built with respect to a `nil` dom record. If it were possible to test in the module code for a domain being nil, and build a null array as a result, this difficulty could be avoided in an obvious fashion. On the other hand, zero is not necessarily a valid initial value for an arbitrary user-defined type; thus, it is more desirable to leave the first initialization to the semantics of the initialization phase.

The provision in the current proposal for zero-initialization only if an explicit [argument or default value] initializer is not supplied handles this problem in the same manner as the existing implementation. That is,

it avoids zero initialization if a default value is supplied. Backward compatibility in this respect ought to allow revised constructors to work correctly with the existing implementation of reference counting, record wrapping and runtime type simulation.

1.1.4.8 Conclusion

The proposed new constructor syntax and semantics can be implemented in such a way that it is compatible with current module code, and will thus support the current specialized way of handling domain, array and distribution construction. Suggestions were made regarding how the specialized handling of these types can be retired incrementally in the future. Each of the categories of reference counting, record-wrapped types and "runtime type" handling can be treated separately from the new constructor story.