1 Classes

[Sections unrelated to constructors have been omitted.]

1.1 Constructors

A class constructor is a special method bound to a class.[1] In addition to the capabilities of a normal method, constructors have an initialization clause which can call a base-class constructor and can initialize member fields. This section describes constructors in terms of their declaration syntax, invocation syntax and execution semantics.

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 §1.1.1.1.

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*$_{opt}$ *initialization−clause constructor−body*
>
> *constructor−name*:
>   *identifier*
>
> *initialization−clause*:
>   *initialization*
>   { *base−class−initializer*$_{opt}$ *initialization−list*$_{opt}$ }
>
> *base−class−initializer*:
>   **super** *argument−list* ;
>   **super** . *constructor−name argument−list* ;
>
> *initialization−list*:
>   *initialization*
>   *initialization−list* ; *initialization*
>
> *initialization*:
>   *expression*

[1] We expect record constructors to be defined similarly.

Aside from the `constructor` keyword, the syntax elements of a constructor including *linkage-specifier*, *type-binding* and *where-clause* have the same syntax as for a method declaration. In contrast, a constructor declaration has two bodies associated with it. The first is an *initialization-clause* and the second is the *constructor-body*. Statements within a *constructor-body* have the same syntax as a *function-body*. An *initialization-clause* has the same syntax as a *function-body* with some restrictions:

- Only a *base-class-initializer* call and/or initialization statements may appear within an *initialization-clause*.
- If present, the *base-class-initializer* call must appear before any other statement.
- Method invocations which would access the object being constructed (i.e. `this`) are disallowed. This restriction includes accessor functions.
- The assignment operator = means initialization, not assignment. That is, assignment semantics are not performed.
- Runtime conditionals are disallowed; conditionals involving `param` and `type` variables are permitted.

In the context of an *initialization-clause*, all references to field names access the contents of that field directly. That is, neither compiler-generated nor user-supplied accessor functions are invoked. Assignment to a field is equivalent to initialization. Reading the value out of a field returns the raw data contained therein, without causing any side-effects.

Within an *initializer-clause*, the `this` keyword may be used to disambiguate field names from argument names. However, in that context it 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 the *initialization-clause*. In particular, it is permissible to call methods on fields within the object being constructed; only calls to methods on the object as a whole (`this`) are prohibited. Thus, the code required to initialize a sync variable or the elements of an array can be placed within the *intialization-clause*.

Within an *initialization-clause*, conditional expressions depending on a run-time value are disallowed.[2] Conditional expressions depending on `param` or `type` expressions are permitted. Similarly, normal `for` and `forall` loops are disallowed, but loops depending on a `param` index are permitted. Parallel constructs are permitted.[3]

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.

[2]We may choose to relax this at a later date, leaving it up to the compiler to perform flow analysis and ensure that a given field has been initialized on every possible path. If a field is initialized along one path but not another, the initialization needs to be introduced only onto the path not containing an initialization, so that explicit initializations are not overridden.

[3]Initially, I put a synchronization point at the point of initialization. However, analysis of data dependencies can span the *initialization-clause* and *constructor-body*. Therefore, the implicit synchronization has been removed.

```
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.
  }
  {}
}
```

Because they do not reference the object being created, static methods may be invoked within an *initialization−clause*.

Rationale. The *constructor−body* is supplied as a place to perform common post-initialization steps. Since execution of the *constructor−body* follows the initialization point, the same steps could be performed at the call site (i.e. where the new expression appears). However, good coding practice would suggest migrating these common steps into the constructor itself.

Suppose, for example, that we wish to increment a count each time one of the above `Point` objects is created. If constructors do not contain a body, it is the responsibility of the caller to increment this count.

```
var pointCount = 0;

constructor Point.__init__(a:real = 0.0, b:real = 0.0)
{ x = a; y = b; }
{ /* No constructor body. */ }

var p0 = new Point(0.0, 0.0);
pointCount += 1;
var p1 = new Point(0.0, 1.0);
pointCount += 1;
var p2 = new Point(1.0, 0.0);
pointCount += 1;
var p3 = new Point(1.0, 1.0);
pointCount += 1;
// Very cumbersome!
```

On the other hand, if constructor bodies are allowed, we can have:

```
var pointCount = 0;

constructor Point.__init__(a:real = 0.0, b:real = 0.0)
{ x = a; y = b; }
{ pointCount += 1; }

var p0 = new Point(0.0, 0.0);
var p1 = new Point(0.0, 1.0);
var p2 = new Point(1.0, 0.0);
var p3 = new Point(1.0, 1.0);
// Not so bad....
```

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 — would permit 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 would be to select a stereotyped name (for example `__init__`) which would be implied if only the class name were 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`.

1.1.1.1 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 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. The arguments appear in the order in which the correponding fields appear in the class declaration (i.e. in declaration order).

The *initialization-clause* 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 *initialization-clause* 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; }
{}
```

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 call (i.e. a `new` expression).

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 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 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 or a resolution error can occur.

1.1.1.2 Default Constructors

As in C++, the default constructor for a class is a constructor 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.

1.1.2 Constructor Invocation

A constructor is invoked using the `new` operator. The rest of the expression looks like a static method call, supplying the name of the constructor (qualified by the class name) and a list of constructor arguments. Constructor invocations do not have a parentheses-less form. If the constructor name is omitted, the default constructor name `__init__` is used. The usual function resolution rules (§**??**) are applied to select among overloaded constructor declarations.

constructor-call-expression:
  new *class-name . constructor-name argument-list*
  new *class-name argument-list*

class-name:
  identifier

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 semantics of a constructor are the actions which occur as a result of a constructor invocation. These can be conceptually divided into four phases: allocation, explicit initialization, default initialization and construction. *Allocation* reserves enough memory to represent the object instance; *explicit initialization* is comprised of the statments in the *initialization-clause*; *default initialization* provides default values for fields which are not explicitly initialized in the *initialization-clause*; and *construction* consists of the statements contained in the *constructor−body*. The implied ordering[4] of these phases is: allocation, then explicit initialization, then default initialization, then construction. We define the *initialization point* as lying after default initialization is complete and before construction has begun.

Allocation consists of reserving on the heap (managed by the current locale) sufficient memory to represent the object being created. This quantity includes all of the non-`param` and non-`type` fields declared in the most-derived class, plus all of those declared in a base-class (if this a derived class) and so on, recursively.

Explicit initialization consists of the statements within the initialization clause. The initialization clause must not contain run-time iteration or conditionals. That is, it must resolve to straight-line code.[5] Aside from that restriction, it is quite flexible in its content. It not only allows the constructor author to rename arguments, but allows routines visible within the scope of the *initialization−clause* to be invoked. The initializer for a given field may depend on the initial values of other fields, and the given field may be initialized multiple times. Invocations of methods of this class on *other instances* of the class is permitted; only method invocations on *this* instance are prohibited — for the reason that this instance has not yet reached its initialization point.

Default initialization provides initial values for fields which are not explicitly initialized in the *initialization−clause* as follows: The compiler creates a list of all of the fields defined in this class (excluding base-class fields). It then scans the *initialization−clause* and removes from the list any fields which appear on the left side of an initialization expression at least once. It is an error for any field to be read (i.e. used as an rvalue in an expression, or as an argument (except one having `out` intent) before it has been explicitly initialized. A default initializer is created for each field remaining on the list. Default initializers are added to the constructor implementation in the order in which the corresponding fields appear in the class declaration (i.e. in declaration order). A default initializer moves a default value into the corresponding field. If a default value is provided as part of the field declaration, that value is used. Otherwise, a type-dependent default value is used (see §**??**).

If the class is a derived class, the *initialization−clause* is also checked for the presence of a base-class constructor call. If none is present, then a call to the default base-class constructor is added implicitly. Whether invoked implicitly or explicitly, the base-class constructor runs to completion — executing both the initialization phases and the construction phase — before returning. Note that the allocation phase is actually part of the execution of the `new` operator; it is not repeated when a base-class constructor is called. Any methods invoked on `this` in the *constructor−body* of a base-class constructor are dispatched as if the run-time type of the object is that of the base class, not of the most-derived class. This prevents a virtual method in the most-derived class from being invoked before the object has reached its initialization point. Because the base-class constructor call must always appear first in the initialization clause, the base-class subobject reaches its initialization point before any of the fields declared in the derived class is initialized.

1.1.3.1 Guaranteed Initialization

A consequence of the default initialization phase and the automatic insertion of missing initializers by the compiler is that by the time execution reaches the initialization point, every field in the object has been initialized to a known value. This property is known as *guaranteed initialization*, and it supplies an important semantic meaning to the syntactical gap between the *initialization−clause* and the *constructor−body*.

[4] I intend to insert some verbiage elsewhere in the Chapel spec indicating that the compiler is free to reorder or execute in parallel statements which have no implied temporal dependencies. The intention is provide the compiler maximum leeway for optimization, and to guide implementors toward an execution model that is massively parallel and data-driven, while still allowing programmers to reason in terms of an ordered sequence of steps.

[5] This restriction allows the compiler to determine which fields have been initialized explicitly, so it can insert the required default initializations.

1.1.4 Inheriting from Multiple Classes

The current constructor story does not support multiple class inheritance. Single-class, multiple-interface inheritance is not ruled out.

1.1.5 Changes from Existing Behavior

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

1.1.5.1 Syntax

At a high level, the only syntactical change is the addition of an *initialization−clause*. Some refactoring of the language syntax may permit the grammar for *initialization* expressions to be specified as a subset of the most general *expression* nonterminal. If so, then the restriction forcing the *initialization−clause* to be straight-line code can be specified within the syntax description, and adherence to be checked during parsing. In any case, 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 *initialization−clause* 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 body of the constructor is called.
- 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.

1.1.5.2 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.

1.1.5.3 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.

1.1.5.4 Default Initialization

Default initialization is also unchanged from the current implementation, until the effect of the initialization clause is considered. In the current implementation, the point of initialization occurs in the default constructor wrapper, just before the constructor itself is called.[6]

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−clause* is empty, the effect will be the same as for the current implementation, except that the body 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−clause* essentially performs the same function as the default wrapper in the current implementation.

However, the addition of the *initialization−clause* 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: first avoiding initializer clauses (current implementation), and then using them (proposed implementation).

```
// 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.
```

[6]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.

```
      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;
    }
  }

  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;
  }


  // 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.

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

constructor RectPoint.RectPoint(p:Point)
{ _x = p.x(); _y = p.y(); }
  // Fully-constructed here.
{ /* Other stuff */ }

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

In this case, each constructor puts the object in a consistent state by the time the end of the constructor clause 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 in the initialization clause. 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.

1.1.5.5 Construction

Construction is basically the same in the existing and proposed schemes. Both assume that the object is fully-constructed by the time the constructor body is entered. However, 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. This is an apparent weakness built into the current version of the language.

It is also important to note that assignment within the body of a constructor 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.[7] 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.

1.1.5.6 Inheritance

Base-class constructor calls are supported in the proposed implementation. The currently proposed semantics are that the base-class constructor must appear before any field initializers in the initialization clause of the derived-class constructor. We may be able to relax this a bit, by stressing that a field cannot be used before it has been initialized. This rule would also apply to the base-class subobject, treating it as a unit. Rules regarding initialization ordering among fields would also treat the base-class "field" uniformly.

Since every constructor is supposed to provide guaranteed initialization, a call to a base-class constructor provides guaranteed initialization of the base-class subobject. 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.

1.1.5.7 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.

[7] 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 its 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) merely 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 this behavior. 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 ¡=¿ 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 ultimately 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 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 must 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.

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 need be implemented as part of the constructor itself — or altnatively 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 (I). t 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 must be 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__convertRuntime TypeToValue()` 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 are apparently 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. 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 worked around 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 clause.

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.

1.1.5.8 Conclusion

It appears that 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 in the preceding section, 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.