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.

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.

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:
  statement
  { *base−class−initializer*$_{opt}$ *statement−list*$_{opt}$ }

base−class−initializer:
  super *argument−list* ;
  super . __**init**__ *argument−list* ;

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, however, 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:

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

- An *initialization−clause* may contain a *base−class−initializer* call. If present, the *base−class−initializer* call must appear before any other statments.
- Within the *initialization−clause*, 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.

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.

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.
  }
  {}
}
```

Because they do not reference any object (and most particularly do not referenct 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 wished to increment a count each time one of the above `Point` objects was created. If constructors did not contain a body, it would be the responsibility of the caller to increment this count.

```
var pointCount = 0;

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, 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;

  constructor byLocation(x: real, y: real)
  { this.x = x; this.y = y; this.message = "a point"; }
  {}

  constructor byMessage(message: string)
  { this.x = 0; this.y = 0; 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 the second constructor lets the user specify the initial message when creating a `MessagePoint`.

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 either by the field initializer in the class declaration, if present, or 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 lexical 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 lexical order). The body of the compiler-generated constructor contains a call to the `initialize` method for the class if one is defined; otherwise, it is empty. 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 lexical order — and so on. Any ancestor field name which is shadowed by 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.

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 default value `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.

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` argument which are handled differently and may need to satisfy additional requirements. See Section **??** for 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*; *construction* consists of the statements contained in the *constructor−body*. The implied ordering[2] 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 is quite flexible. It not only allows the constructor author to rename arguments, but allows routines visible within the scope of the *initialization−clause* to be invoked. 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 lexical 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 of the initialization phases and the construction phase — before returning. 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 always appears 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.

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 initialization point.

[2] 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.

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.