Transitioning from Constructors to Initializers in Chapel

Lydia Duncan (Cray Inc, presenting author) and Michael Noakes (Cray Inc)

Chapel is transitioning from the use of constructors for class and record types to a new strategy called initializers. Initializers retain the convenience of constructors for simple types and provide additional power and flexibility for more complex types. This talk will introduce the core concepts, describe extensions that enhance flexibility, and justify key design choices. These features will be discussed using examples based on Chapel library code.

Initializing an instance has three steps; allocate memory, initialize each field, and perform any additional steps necessary to prepare the instance for use. For simple types the compiler can generate a suitable initializer from the declaration of the type's fields. When customization is required, a type designer can define one or more **init** methods and a **postinit** method. The specification for these methods includes policies that promote safety and convenience.

An **init** method is used to initialize fields. Chapel's approach to generic types and value dependent types requires an **init** method for a derived class to delegate to an **init** method for the parent class before any local fields are initialized. The type designer may rely on the compiler to insert a call to **super.init**() at the start of the method when appropriate. Local fields must be initialized in field declaration order. This policy allows the compiler to insert default initializations on a field by field basis.

An **init** method may call non-**init** methods and may pass **this** as an actual when calling a function. For records, every field must be initialized before it is safe to perform either of these operations. The requirements for class types are more complex. An instance of a class type is not fully initialized until the **init** method for the most derived class has completed, and method calls dispatch on the dynamic type of the object. This requires the compiler to refine the dynamic type of the instance during the execution of the **init** methods. This talk will provide examples that clarify these policies.

The **postinit** method is invoked when the designated **init** method returns. This method provides an opportunity to finalize an instance when the compiler generated **init** method is sufficient to initialize the fields. For instances that are defined by a hierarchy of class declarations, any method call within any definition of **postinit** will be based on the instance's actual dynamic type.

The talk will conclude with the current status of the implementation, potential extensions, and design choices that may be revisited as users gain experience.