SANDIA REPORT

SAND20XX-XXXX
Unlimited Release
Printed January 20XX

VTK-m Users' Guide
Version 0.0

Kenneth Moreland

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation,
a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's
National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND20XX-XXXX
Unlimited Release
Printed January 20XX

VTK-m Users' Guide

Version 0.0

Kenneth Moreland
Scalable Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1323
Albuquerque, NM 87185-1323
kmorel@sandia.gov

Abstract

[Write this.]

Acknowledgement

[Write this. Can steal from Dax document.]

Contents

List of Figures

List of Examples

Chapter 1

Introduction

[Write this. Combination of chapters 1 and 2 in the Dax report plus added features like datasets and filters.]

[Also include the following.]

VTK-m is written in C++ and makes extensive use of templates. The toolkit is implemented as a header library, meaning that all the code is implemented in header files (with extension .h) and completely included in any code that uses it. [Verify that does not change by version 1.0] This is typically necessary of template libraries, which need to be compiled with template parameters that are not known until they are used. This also provides the convenience of allowing the compiler to inline user code for better performance.

When documenting the VTK-m API, the following conventions are used.

- Filenames are printed in a sans serif font.
- C++ code is printed in a `monospace font`.
- Macros and namespaces from the Dax toolkit are printed in `red`.
- Identifiers from the Dax toolkit are printed in `blue`.
- Signatures, described in Section ??, and the tags used in them are printed in `green`.

Chapter 2

Basic Provisions

This section describes the core facilities provided by VTK-m. These include macros, types, and classes that define the environment in which code is run, the core types of data stored, and template introspection. We also start with a description of package structure used by VTK-m.

2.1 Package Structure

VTK-m is organized in a hierarchy of nested packages. VTK-m places definitions in *namespaces* that correspond to the package (with the exception that one package may specialize a template defined in a different namespace).

The base package is named `vtkm`. All classes within VTK-m are placed either directly in the `vtkm` package or in a package beneath it. This helps prevent name collisions between VTK-m and any other library.

As described in Section **??**, the VTK-m API is divided into two distinct environments: the control environment and the execution environment. The API for these two environments are located in the `vtkm::cont` and `vtkm::exec` packages, respectively. Items located in the base `vtkm` namespace are available in both environments.

Although it is conventional to spell out names in identifiers (see the coding conventions in Chapter 11), there is an exception to abbreviate control and execution to `cont` and `exec`, respectively. This is because it is also part of the coding convention to declare the entire namespace when using an identifier that is part of the corresponding package. The shorter names make the identifiers easier to read, faster to type, and more feasible to pack lines in 80 column displays. These abbreviations are also used instead of more common abbreviations (e.g. ctrl for control) because, as part of actual English words, they are easier to type.

[Probably put a paragraph on filters here and move this paragraph lower.]

Worklets provided by VTK-m, described in Chapter **??**, are contained in the `vtkm::-worklet` package. Although the operation of a worklet happens exclusively in the execution

[MAKE A GRAPHIC OF THE PACKAGE HIERARCHY.]

Figure 2.1: VTK-m package hierarchy.

environment, worklets are typically initialized in the control environment. Thus, the `vtkm::worklet` package is not encapsulated in either `vtkm::cont` or `vtkm::exec`.

VTK-m provides a base set of library functions that are ported to the various systems and compilers on which it is used. These functions are located in the `vtkm::math` package. The features in `vtkm::math` are available in both the control and execution environments, but they are typically used in the execution environment.

VTK-m contains code that uses specialized compiler features, such as those with CUDA and OpenMP, or libraries, such as Intel Threading Building Blocks, that will not be available on all machines. Code for these features are encapsulated in their own packages: `vtkm::cuda`, `vtkm::openmp`, and `vtkm::tbb`. Within each one of these packages, there will be `cont` and `exec` namespaces as necessary to denote features that are accessible in only one environment or the other. [I'M THINKING OF REVERSING THIS TO PUT THE DEVICE-SPECIFIC NAMESPACES UNDER `VTKM::CONT`. WE HAVE YET TO NEED A DEVICE-SPECIFIC THING IN `VTKM::EXEC`.]

VTK-m contains OpenGL interoperability that allows data generated with VTK-m to be efficiently transferred to OpenGL objects. This feature is encapsulated in the `vtkm::opengl` package.

Figure 2.1 provides a diagram of the VTK-m package hierarchy.

By convention all classes will be defined in a file with the same name as the class name (with a .h extension) located in a directory corresponding to the package name. For example, the `vtkm::cont::ArrayHandle` class is found in the vtkm/cont/ArrayHandle.h header. There are, however, exceptions to this rule. Some smaller classes and types are grouped together for convenience. These exceptions will be noted as necessary.

Within each namespace there may also be `internal` and `detail` sub-namespaces. The `internal` namespaces contain features that are used internally and may change without notice. The `detail` namespaces contain features that are used by a particular class but must be declared outside of that class. Users should generally ignore classes in these namespaces.

2.2 Function and Method Exports

Any function or method defined by VTK-m must come with an export modifier that determines in which environments the function may be run. These export modifiers are C macros that VTK-m uses to instruct the compiler for which architectures to compile each method. Most user code outside of VTK-m need not use these macros with the important exception of any classes passed to VTK-m. This occurs when defining new worklets, array storage, and device adapters.

VTK-m provides three export macros, VTKM_CONT_EXPORT, VTKM_EXEC_EXPORT, and VTKM_EXEC_CONT_EXPORT, which are used to declare functions and methods that can run in the control environment, execution environment, and both environments, respectively. These macros get defined by including just about any VTK-m header file, but including vtkm/Types.h will ensure they are defined.

The export macro is place after the template declaration, if there is one, and before the return type for the function. Here is a simple example of a function that will square a value. Since most types you would use this function on have operators in both the control and execution environments, the function is exported to both places.

Example 2.1: Usage of export macro.

```
template<typename ValueType>
VTKM_EXEC_CONT_EXPORT
ValueType Square(const ValueType &inValue)
{
  return inValue * inValue;
}
```

The primary function of the export macros is to inject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control exports have `__host__` in them and execution exports have `__device__` in them.

There is one additional export macro that is not used for functions but rather used when declaring a constant data object that is used in the execution environment. This macro is named VTKM_EXEC_CONSTANT_EXPORT and is used to declare a constant lookup table used when executing a worklet. Its primary reason for existing is to add a `__constant__` keyword when compiling with CUDA. This export currently has no effect on any other compiler.

Finally, it is sometimes the case that a function declared as VTKM_EXEC_CONT_EXPORT has to call a method declared as VTKM_EXEC_EXPORT or VTKM_CONT_EXPORT. Generally functions should not call other functions with incompatible control/execution exports, but sometimes a generic VTKM_EXEC_CONT_EXPORT function calls another function determined by the template parameters, and the export of this subfunction may be inconsistent. For cases like this, you can use the VTKM_SUPPRESS_EXEC_WARNINGS to tell the compiler to ignore the inconsistency when resolving the template. When applied to a templated function or method, VTKM_SUPPRESS_EXEC_WARNINGS is placed before the `template` keyword. When applied to a non-templated method in a templated class, VTKM_SUPPRESS_EXEC_WARNINGS is placed before the export macro.

2.3 Core Data Types

Except in rare circumstances where precision is not a concern, VTK-m does not directly use the core C types like `int`, `float`, and `double`. Instead, VTK-m provides its own core types, which are declared in vtkm/Types.h.

2.3.1 Single Number Types

To ensure portability across different compilers and architectures, VTK-m provides `typedef`s for the following basic types with explicit precision: `vtkm::Float32`, `vtkm::Float64`, `vtkm::Int8`, `vtkm::Int16`, `vtkm::Int32`, `vtkm::Int64`, `vtkm::UInt8`, `vtkm::UInt16`, `vtkm::UInt32`, and `vtkm::UInt64`. Under most circumstances when using VTK-m (and performing visualization in general) the type of data is determined by the source of the data or resolved through templates. In the case where a specific type of data is required, these VTK-m–defined types should be preferred over basic C types like `int` or `float`.

Many of the structures in VTK-m require indices to identify elements like points and cells. All indices for arrays and other lists use the type `vtkm::Id`. By default this type is a 32-bit wide integer but can be easily changed by compile options. The CMake configuration option VTKM_USE_64BIT_IDS can be used to change `vtkm::Id` to be 64 bits wide. This configuration can be overridden by defining the C macro `VTKM_USE_64BIT_IDS` or `VTKM_NO_64BIT_IDS` to force `vtkm::Id` to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

There is also a secondary index type named `vtkm::IdComponent` that is used to index components of short vectors (discussed in Section 2.3.2). This type is an integer that might be a shorter width than `vtkm::Id`.

There is also the rare circumstance in which an algorithm in VTK-m computes data values for which there is no indication what the precision should be. For these circumstances, the type `vtkm::FloatDefault` is provided. By default this type is a 32-bit wide floating point number but can be easily changed by compile options. The CMake configuration option VTKM_USE_DOUBLE_PRECISION can be used to change `vtkm::FloatDefault` to be 64 bits wide. This configuration can be overridden by defining the C macro `VTKM_USE_DOUBLE_PRECISION` or `VTKM_NO_DOUBLE_PRECISION` to force `vtkm::FloatDefault` to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

For convenience, you can include either vtkm/internal/ConfigureFor32.h or vtkm/internal/ConfigureFor64.h to force both `vtkm::Id` and `vtkm::FloatDefault` to be 32 or 64 bits.

2.3.2 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides the vtkm::Vec<T,Size> templated type, which is essentially a fixed length array of a given type.

The default constructor of vtkm::Vec objects leaves the values uninitialized. All vectors have a constructor with one argument that is used to initialize all components. All vtkm::Vec objects with a size of 4 or less is specialized to also have a constructor that allows you to set the individual components. Likewise, there is a vtkm::make_Vec function that builds initialized vector types of up to 4 components. Once created, you can use the bracket operator to get and set component values with the same syntax as an array.

Example 2.2: Creating vector types.

```
vtkm::Vec<vtkm::Float32,3> A(1);                          // A is (1, 1, 1)
A[1] = 2;                                                 // A is now (1, 2, 1)
vtkm::Vec<vtkm::Float32,3> B(1, 2, 3);                    // B is (1, 2, 3)
vtkm::Vec<vtkm::Float32,3> C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
```

The types vtkm::Id2 and vtkm::Id3 are typedefs of vtkm::Vec<vtkm::Id,2> and vtkm::Vec<vtkm::Id,2>. These are used to index arrays of 2 and 3 dimensions, which is common.

Vectors longer than 4 are also supported, but independent component values must be set after construction. The vtkm::Vec class contains a constant named NUM_COMPONENTS to specify how many components are in the vector.

Example 2.3: A Longer Vector.

```
vtkm::Vec<vtkm::Float32, 5> A(2); // A is (2, 2, 2, 2, 2)
for (vtkm::IdComponent index = 1; index < A.NUM_COMPONENTS; index++)
{
  A[index] = A[index-1] * 1.5;
}
// A is now (2, 3, 4.5, 6.75, 10.125)
```

vtkm::Vec supports component-wise arithmetic using the operators for plus (+), minus (-), multiply (*), and divide (/). It also supports scalar to vector multiplication with the multiply operator. The comparison operators equal (==) is true if every pair of corresponding components are true and not equal (!=) is true otherwise. A special vtkm::dot function is overloaded to provide a dot product for every type of vector.

Example 2.4: Vector operations.

```
vtkm::Vec<vtkm::Float32,3> A(1, 2, 3);
vtkm::Vec<vtkm::Float32,3> B(4, 5, 6.5);
vtkm::Vec<vtkm::Float32,3> C = A + B;          // C is (5, 7, 9.5)
vtkm::Vec<vtkm::Float32,3> D = 2.0f * C;       // D is (10, 14, 19)
vtkm::Float32 s = vtkm::dot(A, B);             // s is 33.5
bool b1 = (A == B);                            // b1 is false
bool b2 = (A == vtkm::make_Vec(1, 2, 3));      // b2 is true
```

These operators, of course, only work if they are also defined for the component type of the vtkm::Vec. For example, the multiply operator will work fine on objects of type vtkm::Vec<char,3>, but the multiply operator will not work on objects of type vtkm::-Vec<std::string,3> because you cannot multiply objects of type std::string.

In addition to generalizing vector operations and making arbitrarily long vectors, vtkm::-Vec can be repurposed for creating any sequence of homogeneous objects. Here is a simple example of using vtkm::Vec to hold the state of a polygon.

Example 2.5: Repurposing a vtkm::Vec.

```
vtkm::Vec<vtkm::Vec<vtkm::Float32,2>, 3> equilateralTriangle(
                                          vtkm::make_Vec(0.0, 0.0),
                                          vtkm::make_Vec(1.0, 0.0),
                                          vtkm::make_Vec(0.5, 0.866));
```

2.3.3 Extents

vtkm::Extent3 is a simple structure that holds the extent information for structured data (data defined on a regular grid). It contains two vtkm::Id3 fields named Min and Max that define the minimum and maximum 3D index. Min and Max are *inclusive* point indices.

Although less used, there also exists vtkm::Extent2, which is the same as vtkm::Extent3 except for structured grids with 2 topological dimensions. The two structures are exactly the same except that vtkm::Extent2 uses vtkm::Id2 objects for its Min and Max instead of vtkm::Id3. There is also a generic structure named vtkm::Extent that takes a single integer argument to make the structured extent for some arbitrary topological dimension.

vtkm::Extent3 and the other extent structures are defined in the vtkm/Extent.h header. This header also contains the following helper functions.

vtkm::ExtentPointDimensions Takes an extent object as an argument and returns a vtkm::Id3 (or other appropriately sized tuple) giving the number of points in each topological dimension.

vtkm::ExtentCellDimensions Takes an extent object as an argument and returns a vtkm::Id3 (or other appropriately sized tuple) giving the number of cells in each topological dimension. The number of cells is one less than the number of points in each dimensions.

vtkm::ExtentNumberOfPoints Takes an extent object as an argument and returns the number of points in an associated structured mesh.

vtkm::ExtentNumberOfCells Takes an extent object as an argument and returns the number of cells in an associated structured mesh.

vtkm::ExtentPointFlatIndexToTopologyIndex Elements in structured grids have a single index with 0 being the entry at the minimum extent in every direction and then increasing first in the r direction, then the s direction, and then the t direction. This function takes a flat point index as its first argument and an extent as its second argument and returns the r, s, t topological coordinates in a `vtkm::Id3` (or other appropriately sized tuple).

vtkm::ExtentCellFlatIndexToTopologyIndex Takes a flat cell index as its first argument and an extent as its second argument and returns the r, s, t topological coordinates in a `vtkm::Id3` (or other appropriately sized tuple).

vtkm::ExtentPointTopologyIndexToFlatIndex Takes topological point coordinates as the first argument and an extent as the second argument and returns the equivalent flat point index.

vtkm::ExtentCellTopologyIndexToFlatIndex Takes topological cell coordinates as the first argument and an extent as the second argument and returns the equivalent flat cell index.

vtkm::ExtentFirstPointOnCell Takes a flat cell index as its first argument and returns the flat index to the first point incident on that cell. This is a convenience function for some operations that relate cells to points.

The following example demonstrates using a `vtkm::Extent3` and the supporting functions. Extents of different dimensions work corredspondingly.

Example 2.6: Creating and using an `Extent3`.

```
#include <vtkm/Extent.h>
#include <vtkm/Types.h>

void ExtentExample()
{
  // Make an extent that defines a grid that has 5x5x3 points and "centered"
  // at index (0,0,0).
  vtkm::Extent3 extent(vtkm::Id3(-2,-2,-1), vtkm::Id3(2,2,1));

  vtkm::Id3 minIndices = extent.Min; // Is (-2,-2,-1)
  vtkm::Id3 maxIndices = extent.Max; // Is (2,2,1)

  vtkm::Id3 pointDims = vtkm::ExtentPointDimensions(extent); // Returns (5,5,3)
  vtkm::Id3 cellDims = vtkm::ExtentCellDimensions(extent);   // Returns (4,4,2)

  vtkm::Id numPoints = vtkm::ExtentNumberOfPoints(extent); // Returns 75
  vtkm::Id numCells = vtkm::ExtentNumberOfCells(extent);   // Returns 32

  // Returns (-1,-1,0)
  vtkm::Id3 pointIndexA = vtkm::ExtentPointFlatIndexToTopologyIndex(31, extent);

  // Returns (1,1,0)
  vtkm::Id3 cellIndexA = vtkm::ExtentCellFlatIndexToTopologyIndex(31, extent);

  // Returns 33
  vtkm::Id pointIndexB =
      vtkm::ExtentPointTopologyIndexToFlatIndex(vtkm::Id3(1,-1,0), extent);
  // Returns 23
  vtkm::Id cellIndexB =
      vtkm::ExtentCellTopologyIndexToFlatIndex(vtkm::Id3(1,-1,0), extent);

  // Returns 33
  vtkm::Id firstPoint = vtkm::ExtentFirstPointOnCell(23, extent);
}
```

2.3.4 Pair

VTK-m defines a `vtkm::Pair<T1,T2>` templated object that behaves just like `std::pair` from the standard template library. The difference is that `vtkm::Pair` will work in both the execution and control environment, whereas the STL `std::pair` does not always work in the execution environment.

The VTK-m version of `vtkm::Pair` supports the same types, fields, and operations as the STL version. VTK-m also provides a `vtkm::make_Pair` function for convenience.

2.4 Traits

When using templated types, it is often necessary to get information about the type or specialize code based on general properties of the type. VTK-m uses traits classes to publish and retrieve information about types. A traits class is simply a templated structure that provides typedefs for tag structures, empty types used for identification. The traits classes might also contain constant numbers and helpful static functions. See *Effective C++ Third Edition* by Scott Mayers for a description of traits classes and their uses.

2.4.1 Type Traits

The `vtkm::TypeTraits<T>` templated class provides basic information about a core type. These type traits are available for all the basic C++ types as well as the core VTK-m types described in Section 2.3. `vtkm::TypeTraits` contains the following elements.

`NumericTag` This type is set to either `vtkm::TypeTraitsRealTag` or `vtkm::TypeTraitsIntegerTag` to signal that the type represents either floating point numbers or integers.

`DimensionalityTag` This type is set to either `vtkm::TypeTraitsScalarTag` or `vtkm::TypeTraitsVectorTag` to signal that the type represents either a single scalar value or a tuple of values.

The definition of `vtkm::TypeTraits` for `vtkm::Float32` could like something like this.

Example 2.7: Definition of vtkm::TypeTraits<vtkm::Float32>.

```
namespace vtkm {

template<>
struct TypeTraits<vtkm::Float32>
{
  typedef vtkm::TypeTraitsRealTag NumericTag;
  typedef vtkm::TypeTraitsScalarTag DimensionalityTag;
};

}
```

Here is a simple example of using vtkm::TypeTraits to implement a generic function that behaves like the remainder operator (%) for all types including floating points and vectors.

Example 2.8: Using TypeTraits for a generic remainder.

```
#include <vtkm/TypeTraits.h>

#include <vtkm/Math.h>

template<typename T>
T Remainder(const T &numerator, const T &denominator);

namespace detail {

template<typename T>
T RemainderImpl(const T &numerator,
                const T &denominator,
                vtkm::TypeTraitsIntegerTag,
                vtkm::TypeTraitsScalarTag)
{
  return numerator % denominator;
}

template<typename T>
T RemainderImpl(const T &numerator,
                const T &denominator,
                vtkm::TypeTraitsRealTag,
                vtkm::TypeTraitsScalarTag)
{
  // The VTK-m math library contains a Remainder function that operates on
  // floating point numbers.
  return vtkm::Remainder(numerator, denominator);
}

template<typename T, typename NumericTag>
T RemainderImpl(const T &numerator,
                const T &denominator,
                NumericTag,
                vtkm::TypeTraitsVectorTag)
{
  T result;
  for (int componentIndex = 0;
       componentIndex < T::NUM_COMPONENTS;
       componentIndex++)
  {
    result[componentIndex] =
        Remainder(numerator[componentIndex], denominator[componentIndex]);
  }
  return result;
}

} // namespace detail
```

```
template<typename T>
T Remainder(const T &numerator, const T &denominator)
{
  return detail::RemainderImpl(numerator,
                               denominator,
                               typename vtkm::TypeTraits<T>::NumericTag(),
                               typename vtkm::TypeTraits<T>::DimensionalityTag());
}
```

2.4.2 Vector Traits

The vtkm::VecTraits<T> templated class provides information and accessors to vector types. It contains the following elements.

ComponentType This type is set to the type for each component in the vector. For example, a vtkm::Id3 has ComponentType defined as vtkm::Id.

NUM_COMPONENTS An integer specifying how many components are contained in the vector.

HasMultipleComponents This type is set to either vtkm::VecTraitsTagSingleComponent if the vector length is size 1 or vtkm::VecTraitsTagMultipleComponents otherwise. This tag can be useful for creating specialized functions when a vector is really just a scalar.

GetComponent A static method that takes a vector and returns a particular component.

SetComponent A static method that takes a vector and sets a particular component to a given value.

ToVec A static method that converts a vector of the given type to a vtkm::Vec.

The definition of vtkm::VecTraits for vtkm::Id3 could like something like this.

Example 2.9: Definition of vtkm::VecTraits<vtkm::Id3>.

```
namespace vtkm {

template<>
struct VecTraits<vtkm::Id3>
{
  typedef vtkm::Id ComponentType;
  static const int NUM_COMPONENTS = 3;
  typedef VecTraitsTagMultipleComponents HasMultipleComponents;

  VTKM_EXEC_CONT_EXPORT
  static vtkm::Id &GetComponent(vtkm::Id3 &vector, int component) {
    return vector[component];
  }

  VTKM_EXEC_CONT_EXPORT
  static void SetComponent(vtkm::Id3 &vector, int component, vtkm::Id value) {
    vector[component] = value;
  }

  VTKM_EXEC_CONT_EXPORT
  static vtkm::Vec<vtkm::Id,3> ToTuple(const vtkm::Id3 &vector) {
    return vector;
  }
};

} // namespace vtkm
```

The real power of vector traits is that they simplify creating generic operations on any type that can look like a vector. This includes operations on scalar values as if they were vectors of size one. The following code uses vector traits to simplify the implementation of less functors that define an ordering that can be used for sorting and other operations.

Example 2.10: Using `VecTraits` for less functors.

```
#include <vtkm/VecTraits.h>

// This functor provides a total ordering of vectors. Every compared vector
// will be either less, greater, or equal (assuming all the vector components
// also have a total ordering).
template<typename T>
struct LessTotalOrder
{
  VTKM_EXEC_CONT_EXPORT
  bool operator()(const T &left, const T &right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      typedef typename vtkm::VecTraits<T>::ComponentType ComponentType;
      const ComponentType &leftValue =
          vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType &rightValue =
          vtkm::VecTraits<T>::GetComponent(right, index);
      if (leftValue < rightValue) { return true; }
      if (rightValue < leftValue) { return false; }
    }
    // If we are here, the vectors are equal (or at least equivalent).
    return false;
  }
};

// This functor provides a partial ordering of vectors. It returns true if and
// only if all components satisfy the less operation. It is possible for
// vectors to be neither less, greater, nor equal, but the transitive closure
// is still valid.
template<typename T>
struct LessPartialOrder
{
  VTKM_EXEC_CONT_EXPORT
  bool operator()(const T &left, const T &right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      typedef typename vtkm::VecTraits<T>::ComponentType ComponentType;
      const ComponentType &leftValue =
          vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType &rightValue =
          vtkm::VecTraits<T>::GetComponent(right, index);
      if (!(leftValue < rightValue)) { return false; }
    }
    // If we are here, all components satisfy less than relation.
    return true;
  }
};
```

2.5 List Tags

VTK-m internally uses template metaprogramming, which utilizes the C++ template to run source-generating programs, to customize code to various data and compute platforms. One basic structure often uses with template metaprogramming is a list of class names (also sometimes called a tuple or vector, although both of those names have different meanings in VTK-m).

Many VTK-m users only need predefined lists, such as the type lists specified in Section 2.5.2 or the storage lists specified in Section **??**. Those users can skip most of the details of this section. However, it is sometimes useful to modify lists, create new lists, or operate on lists, and these usages are documented here.

VTK-m uses a tag-based mechanism for defining lists, which differs significantly from lists in many other template metaprogramming libraries such as with `boost::mpl::vector` or `boost::vector`. Rather than enumerating all list entries as template arguments, the list is referenced by a single tag class with a descriptive name. The intention is to make fully resolved types shorter and more readable. (Anyone experienced with template programming knows how insanely long and unreadable types can get in compiler errors and warnings.)

2.5.1 Building List Tags

List tags are constructed in VTK-m by defining a `struct` that publicly inherits from another list tags. The base list tags are defined in the vtkm/ListTag.h header.

The most basic list is defined with `vtkm::ListTagEmpty`. This tag represents an empty list.

`vtkm::ListTagBase<T, ...>` represents a list of the types given as template parameters. `vtkm::ListTagBase` supports a variable number of parameters with the maximum specified by `VTKM_MAX_BASE_LIST`.

Finally, lists can be combined together with `vtkm::ListTagJoin<ListTag1,ListTag2>`, which concatinates two lists together.

The following example demonstrates how to build list tags using these base lists classes. Note first that all the list tags are defined as `struct` rather than `class`. Although these are roughly synonymous in C++, `struct` inheritance is by default public, and public inheritance is important for the list tags to work. Note second that these tags are created by inheritance rather than using `typedef`. Although `typedef` will work, it will lead to much uglier type names defined by the compiler.

Example 2.11: Creating list tags.

```
#include <vtkm/ListTag.h>

// Placeholder classes representing things that might be in a template
// metaprogram list.
class Foo;
class Bar;
class Baz;
class Qux;
class Xyzzy;

// The names of the following tags are indicative of the lists they contain.

struct FooList : vtkm::ListTagBase<Foo> { };

struct FooBarList : vtkm::ListTagBase<Foo,Bar> { };

struct BazQuxXyzzyList : vtkm::ListTagBase<Baz,Qux,Xyzzy> { };

struct QuxBazBarFooList : vtkm::ListTagBase<Qux,Baz,Bar,Foo> { };

struct FooBarBazQuxXyzzyList
    : vtkm::ListTagJoin<FooBarList, BazQuxXyzzyList> { };
```

2.5.2 Type Lists

One of the major use cases for template metaprogramming lists in VTK-m is to identify a set of potential data types for arrays. The vtkm/TypeListTag.h header contains predefined lists for known VTK-m types. Although technically all these lists are of C++ types, the types we refer to here are those data types stored in data arrays. The following lists are provided.

`vtkm::TypeListTagId` Contains the single item `vtkm::Id`.

`vtkm::TypeListTagId2` Contains the single item `vtkm::Id2`.

`vtkm::TypeListTagId3` Contains the single item `vtkm::Id3`.

`vtkm::TypeListTagIndex` A list of all types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`.

`vtkm::TypeListTagFieldScalar` A list containing types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`).

`vtkm::TypeListTagFieldVec2` A list containing types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec3` A list containing types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec3` A list containing types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers.

vtkm::TypeListTagField A list containing all the types generally used for fields. It is the combination of vtkm::TypeListTagFieldScalar, vtkm::TypeListTagFieldVec2, vtkm::TypeListTagFieldVec3, and vtkm::TypeListTagFieldVec4.

vtkm::TypeListTagScalarAll A list of all scalar types. It contains signed and unsigned integers of widths from 8 to 64 bits. It also contains floats of 32 and 64 bit widths.

vtkm::TypeListTagVecCommon A list of the most common vector types. It contains all vtkm::Vec class of size 2 through 4 containing components of unsigned bytes, signed 32-bit integers, signed 64-bit integers, 32-bit floats, or 64-bit floats.

vtkm::TypeListTagVecAll A list of all vtkm::Vec classes with standard integers or floating points as components and lengths between 2 and 4.

vtkm::TypeListTagAll A list of all types included in vtkm/Types.h with vtkm::Vecs with up to 4 components.

vtkm::TypeListTagCommon A list containing only the most used types in visualization. This includes signed integers and floats that are 32 or 64 bit. It also includes 3 dimensional vectors of floats. This is the default list used when resolving the type in dynamic arrays (described in Section 6.3).

If these lists are not sufficient, it is possible to build new type lists using the existing type lists and the list bases from Section 2.5.1 as demonstrated in the following example.

Example 2.12: Defining new type lists.

```
#define VTKM_DEFAULT_TYPE_LIST_TAG MyCommonTypes

#include <vtkm/ListTag.h>
#include <vtkm/TypeListTag.h>

// A list of 2D vector types.
struct Vec2List
    : vtkm::ListTagBase<vtkm::Id2,
                        vtkm::Vec<vtkm::Float32,2>,
                        vtkm::Vec<vtkm::Float64,2> > {  };

// An application that uses 2D geometry might commonly encounter this list of
// types.
struct MyCommonTypes : vtkm::ListTagJoin<Vec2List,vtkm::TypeListTagCommon> {  };
```

The vtkm/TypeListTag.h header also defines a macro named VTKM_DEFAULT_TYPE_LIST_TAG that defines a default list of types to use in classes like vtkm::cont::DynamicArrayHandle (Section 6.3). This list can be overridden by defining the VTKM_DEFAULT_TYPE_LIST_TAG macro *before* any VTK-m headers are included. If included after a VTK-m header, the list is not likely to take effect. Do not ignore compiler warnings about the macro being redefined, which you will not get if defined correctly. Example 2.12 also contains an example of overriding the VTKM_DEFAULT_TYPE_LIST_TAG macro.

2.5.3 Operating on Lists

VTK-m template metaprogramming lists are typically just passed to VTK-m methods that internally operate on the lists. Although not typically used outside of the VTK-m library, these operations are also available.

The vtkm/ListTag.h header comes with a `vtkm::ListForEach` function that takes a functor object and a list tag. It then calls the functor object with the default object of each type in the list. This is most typically used with C++ run-time type information to convert a run-time polymorphic object to a statically typed (and possibly inlined) call.

The following example shows a rudimentary version of coverting a dynamically-typed array to a statically-typed array similar to what is done in VTK-m classes like `vtkm::cont::DynamicArrayHandle` (which is documented in Section 6.3).

Example 2.13: Converting dynamic types to static types with `ListForEach`.

```
struct MyArrayBase {
  // A virtual destructor makes sure C++ RTTI will be generated. It also helps
  // ensure subclass destructors are called.
  virtual ~MyArrayBase() { }
};

template<typename T>
struct MyArrayImpl : public MyArrayBase {
  std::vector<T> Array;
};

template<typename T>
void PrefixSum(std::vector<T> &array)
{
  T sum(typename vtkm::VecTraits<T>::ComponentType(0));
  for (typename std::vector<T>::iterator iter = array.begin();
       iter != array.end();
       iter++)
  {
    sum = sum + *iter;
    *iter = sum;
  }
}

struct PrefixSumFunctor {
  MyArrayBase *ArrayPointer;

  PrefixSumFunctor(MyArrayBase *arrayPointer) : ArrayPointer(arrayPointer) { }

  template<typename T>
  void operator()(T) {
    typedef MyArrayImpl<T> ConcreteArrayType;
    ConcreteArrayType *concreteArray =
        dynamic_cast<ConcreteArrayType *>(this->ArrayPointer);
    if (concreteArray != NULL)
    {
      PrefixSum(concreteArray->Array);
    }
  }
};

void DoPrefixSum(MyArrayBase *array)
{
  PrefixSumFunctor functor = PrefixSumFunctor(array);
  vtkm::ListForEach(functor, vtkm::TypeListTagCommon());
}
```

Chapter 3

File I/O

[Write this.]

Chapter 4

Provided Filters

[Write this once some filters are provided. Should also make a reference to using worklets (Chapter 7).]

Chapter 5

Rendering

[Write this once the rendering module is implemented.]

Chapter 6

Control Environment

The control environment is where code interfaces with applications and I/O devices. The associated API is designed for users that want to use VTK-m to analyze their data using provided or supplied worklets. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

Most users of VTK-m will have some interaction with the control environment, for you cannot define data structures or execute any algorithms without it.

6.1 Device Adapter Tag

VTK-m uses a feature called a device adapter to define what type of device will be used to run algorithms. The device adapter encapsulates the device-specific code required to port to various devices. More information on the function of the device adapter is given in Section **??**.

The device adapter is identified by a *device adapter tag*. This tag, which is simply an empty struct type, is used as the template parameter for several classes in the VTK-m control environment and causes these classes to direct their work to a particular device.

There are two ways to select a device adapter. The first is to make a global selection of a default device adapter. The second is to specify a specific device adapter as a template parameter.

6.1.1 Default Device Adapter

A default device adapter tag is specified in vtkm/cont/DeviceAdapter.h (although it can also by specified in many other VTK-m headers via header dependencies). If no other information is given, VTK-m attempts to choose a default device adapter that is a best fit for the system it is compiled on. VTK-m currently select the default device adapter with the following sequence of conditions.

- If the source code is being compiled by CUDA, the CUDA device is used.
- If the CUDA compiler is not being used and the current compiler supports OpenMP, then the OpenMP device is used. [Technically, OpenMP is not yet supported in VTK-m, so this will never actually be picked. But once it is implemented, this will be the chain.]
- If the compiler supports neither CUDA nor OpenMP and VTK-m was configured to use Intel Threading Building Blocks, then that device is used.
- If no parallel device adapters are found, then VTK-m falls back to a serial device.

You can also set the default device adapter specifically by setting the VTKM_DEVICE_ADAPTER macro. This macro must be set *before* including any VTK-m files. You can set VTKM_DEVICE_ADAPTER to any one of the following.

VTKM_DEVICE_ADAPTER_SERIAL Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available.

VTKM_DEVICE_ADAPTER_CUDA Uses a CUDA capable GPU device. For this device to work, VTK-m must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler.

VTKM_DEVICE_ADAPTER_OPENMP Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. [Not currently implemented.]

VTKM_DEVICE_ADAPTER_TBB Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use TBB and the executable must be linked to the TBB library.

These macros provide a useful mechanism for quickly reconfiguring code to run on different devices. The following example shows a typical block of code at the top of a source file that could be used for quick reconfigurations.

Example 6.1: Macros to port VTK-m code among different devices

```
// Uncomment one (and only one) of the following to reconfigure the Dax
// code to use a particular device. Comment them all to automatically pick a
// device.
#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_SERIAL
//#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_CUDA
//#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_OPENMP
//#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_TBB

#include <vtkm/cont/DeviceAdapter.h>
```

The default device adapter can always be overridden by specifying a device adapter tag, as described in the next section. There is actually one more internal default device adapter named `VTKM_DEVICE_ADAPTER_ERROR` that will cause a compile error if any component attempts to use the default device adapter. This feature is most often used in testing code to check when device adapters should be specified.

6.1.2 Specifying Device Adapter Tags

In addition to setting a global default device adapter, it is possible to explicitly set which device adapter to use in any feature provided by VTK-m. This is done by providing a device adapter tag as a template argument to VTK-m templated objects. The following device adapter tags are available in VTK-m.

`vtkm::cont::DeviceAdapterTagSerial` Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available. This tag is defined in vtkm/cont/DeviceAdapterSerial.h.

`vtkm::cont::DeviceAdapterTagCuda` Uses a CUDA capable GPU device. For this device to work, VTK-m must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler. This tag is defined in vtkm/cont/cuda/DeviceAdapterCuda.h.

`vtkm::cont::DeviceAdapterTagOpenMP` Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. This tag is defined in vtkm/openmp/cont/DeviceAdapterOpenMP.h. [NOT CURRENTLY IMPLEMENTED.]

`vtkm::cont::DeviceAdapterTagTBB` Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use TBB and the executable must be linked to the TBB library. This tag is defined in vtkm/cont/tbb/DeviceAdapterTBB.h.

[ADD EXAMPLE. I WOULD LIKE TO HAVE A SIMPLE EXAMPLE INVOLVING A FILTER, BUT THAT INTERFACE IS STILL IN FLUX.]

When structuring your code to always specify a particular device adapter, consider setting the default device adapter (with the `VTKM_DEVICE_ADAPTER` macro) to `VTKM_DEVICE_ADAPTER_ERROR`. This will cause the compiler to produce an error if any object is instantiated with the default device adapter, which checks to make sure the code properly specifies every device adapter parameter.

VTK-m also defines a macro named `VTKM_DEFAULT_DEVICE_ADAPTER_TAG`, which can be used in place of an explicit device adapter tag to use the default tag. This macro is used to create new templates that have template parameters for device adapters that can use the default. The following example has a (rather artificial) declaration of a helper class for executing the elevation filter.

[Add example when filters are settled.]

6.2 Array Handle

An *array handle*, implemented with the `vtkm::cont::ArrayHandle` class, manages an array of data that can be accessed or manipulated by VTK-m algorithms. It is typical to construct an array handle in the control environment to pass data to an algorithm running in the execution environment. It is also typical for an algorithm running in the execution environment to allocate and populate an array handle, which can then be read back in the control environment. It is also possible for an array handle to manage data created by one VTK-m algorithm and passed to another, remaining in the execution environment the whole time and never copied to the control environment.

The array handle may have up to two copies of the array, one for the control environment and one for the execution environment. However, depending on the device and how the array is being used, the array handle will only have one copy when possible. Copies between the environments are implicit and lazy. They are copied only when an operation needs data in an environment where the data is not.

`vtkm::cont::ArrayHandle` behaves like a shared smart pointer in that when the C++ object is copied, each copy holds a reference to the same array. These copies are reference counted so that when all copies of the `vtkm::cont::ArrayHandle` are destroyed, any allocated memory is released.

6.2.1 Creating Array Handles

`vtkm::cont::ArrayHandle` is a templated class with two template parameters. The first template parameter is the only one required and specifies the base type of the entries in the array. The second template parameter specifies the storage used when storing data in the control environment. Storage objects are discussed later in this section, and for now we will use the default value.

Example 6.2: Declaration of the `vtkm::cont::ArrayHandle` templated class.

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
class ArrayHandle;
```

There are multiple ways to create and populate an array handle. The default `vtkm::cont::ArrayHandle` constructor will create an empty array with nothing allocated in either the control or execution environment. This is convenient for creating arrays used as the output for algorithms.

Example 6.3: Creating an ArrayHandle for output data.

```
vtkm::cont::ArrayHandle<vtkm::Float32> outputArray;
```

Constructing an vtkm::cont::ArrayHandle that points to a provided C array or std::vector is straightforward with the vtkm::cont::make_ArrayHandle functions. These functions will make an array handle that points to the array data that you provide.

Example 6.4: Creating an ArrayHandle that points to a provided C array.

```
  vtkm::Float32 dataBuffer[50];
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
      vtkm::cont::make_ArrayHandle(dataBuffer, 50);
```

Example 6.5: Creating an ArrayHandle that points to a provided std::vector.

```
  std::vector<vtkm::Float32> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
      vtkm::cont::make_ArrayHandle(dataBuffer);
```

Be aware that vtkm::cont::make_ArrayHandle makes a shallow pointer copy. This means that if you change or delete the data provided, the internal state of vtkm::cont::-ArrayHandle becomes invalid and undefined behavior can ensue. The most common manifestation of this error happens when a std::vector goes out of scope. This subtle interaction will cause the vtkm::cont::ArrayHandle to point to an unallocated portion of the memory heap. For example, if the code in Example 6.5 where to be placed within a callable function or method, it could cause the vtkm::cont::ArrayHandle to become invalid.

Example 6.6: Invalidating an ArrayHandle by letting the source std::vector leave scope.

```
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
{
  std::vector<vtkm::Float32> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
      vtkm::cont::make_ArrayHandle(dataBuffer);

  return inputArray;
  // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
  // memory. However, inputArray has a pointer to that memory, which becomes an
  // invalid pointer in the returned object. Bad things will happen when the
  // ArrayHandle is used.
}

VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad()
{
  std::vector<vtkm::Float32> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> tmpArray =
      vtkm::cont::make_ArrayHandle(dataBuffer);

  // This copies the data from one ArrayHandle to another (in the execution
  // environment). Although it is an extraneous copy, it is usually pretty fast
  // on a parallel device. Another option is to make sure that the buffer in
  // the std::vector never goes out of scope before all the ArrayHandle
  // references, but this extra step allows the ArrayHandle to manage its own
  // memory and ensure everything is valid.
  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray;
  vtkm::cont::DeviceAdapterAlgorithm<VTKM_DEFAULT_DEVICE_ADAPTER_TAG>::Copy(
      tmpArray, inputArray);

  return inputArray;
  // This is safe.
}
```

6.2.2 Array Portals

An array handle defines auxiliary structures called *array portals* that provide direct access into its data. An array portal is a simple object that is somewhat functionally equivalent to an STL-type iterator, but with a much simpler interface. Array portals can be read-only (const) or read-write and they can be accessible from either the control environment or the execution environment. All these variants have similar interfaces although some features that are not applicable can be left out.

An array portal object contains each of the following:

`ValueType` A `typedef` of the type for each item in the array.

`GetNumberOfValues` A method that returns the number of entries in the array.

`Get` A method that returns the value at a given index.

`Set` A method that changes the value at a given index. This method does not need to exist for read-only (const) array portals.

The following code example defines an array portal for a simple C array of scalar values. This definition has no practical value (it is covered by the more general `vtkm::cont::internal::ArrayPortalFromIterators`), but demonstrates the function of each component.

Example 6.7: A simple array portal implementation.

```
template<typename T>
class SimpleScalarArrayPortal
{
public:
  typedef T ValueType;

  // There is no specification for creating array portals, but they generally
  // need a constructor like this to be practical.
  VTKM_EXEC_CONT_EXPORT
  SimpleScalarArrayPortal(ValueType *array, vtkm::Id numberOfValues)
    : Array(array), NumberOfValues(numberOfValues) {  }
  VTKM_EXEC_CONT_EXPORT
  SimpleScalarArrayPortal() : Array(NULL), NumberOfValues(0) {  }

  VTKM_EXEC_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }

  VTKM_EXEC_CONT_EXPORT
  ValueType Get(vtkm::Id index) const { return this->Array[index]; }

  VTKM_EXEC_CONT_EXPORT
  void Set(vtkm::Id index, ValueType value) const {
    this->Array[index] = value;
  }

private:
  ValueType *Array;
  vtkm::Id NumberOfValues;
};
```

Although array portals are simple to implement and use, and array portals' functionality is similar to iterators, there exists a great deal of code already based on STL iterators and it is often convienient to interface with an array through an iterator rather than an array portal. The `vtkm::cont::ArrayPortalToIterators` class can be used to convert an array portal to an STL-compatible iterator. The class is templated on the array portal type and has a constructor that accepts an instance of the array portal. It contains the following features.

`IteratorType` A `typedef` of an STL-compatible random-access iterator that can provide the same access as the array portal.

`GetBegin` A method that returns an STL-compatible iterator of type `IteratorType` that points to the beginning of the array.

`GetEnd` A method that returns an STL-compatible iterator of type `IteratorType` that points to the end of the array.

Example 6.8: Using `ArrayPortalToIterators`.

```
template<typename PortalType>
VTKM_CONT_EXPORT
std::vector<typename PortalType::ValueType>
CopyArrayPortalToVector(const PortalType &portal)
{
  typedef typename PortalType::ValueType ValueType;
  std::vector<ValueType> result(portal.GetNumberOfValues());

  vtkm::cont::ArrayPortalToIterators<PortalType> iterators(portal);

  std::copy(iterators.GetBegin(), iterators.GetEnd(), result.begin());

  return result;
}
```

As a convenience, vtkm/cont/ArrayPortalToIterators.h also defines a pair of functions named `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd` that each take an array portal as an argument and return a begin and end iterator, respectively.

Example 6.9: Using ArrayPortalToIteratorBegin and ArrayPortalToIteratorEnd.

```
std::vector<vtkm::Float32> myContainer(portal.GetNumberOfValues());

std::copy(vtkm::cont::ArrayPortalToIteratorBegin(portal),
          vtkm::cont::ArrayPortalToIteratorEnd(portal),
          myContainer.begin());
```

vtkm::cont::ArrayHandle contains two typedefs for array portal types that are capable of interfacing with the underlying data in the control environment. These are PortalControl and PortalConstControl, which define read-write and read-only (const) array portals, respectively.

vtkm::cont::ArrayHandle also contains similar typedefs for array portals in the execution environment. Because these types are dependent on the device adapter used for execution, these typedefs are embedded in a templated class named ExecutionTypes. Within ExecutionTypes are the typedefs Portal and PortalConst defining the read-write and read-only (const) array portals, respectively, for the execution environment for the given device adapter tag.

Because vtkm::cont::ArrayHandle is an control environment object, it provides the methods GetPortalControl and GetPortalConstControl to get the associated array portal objects. These methods also have the side effect of refreshing the control environment copy of the data, so this can be a way of synchronizing the data. Be aware that when an vtkm::cont::ArrayHandle is created with a pointer or std::vector, it is put in a read-only mode, and GetPortalControl can fail (although GetPortalConstControl will still work). Also be aware that calling GetPortalControl will invalidate any copy in the execution environment, meaning that any subsequent use will cause the data to be copied back again.

Example 6.10: Using portals from an ArrayHandle.

```
template<typename T>
void SortCheckArrayHandle(vtkm::cont::ArrayHandle<T> arrayHandle)
{
  typedef typename vtkm::cont::ArrayHandle<T>::PortalControl
      PortalType;
  typedef typename vtkm::cont::ArrayHandle<T>::PortalConstControl
      PortalConstType;

  PortalType readwritePortal = arrayHandle.GetPortalControl();
  // This is actually pretty dumb. Sorting would be generally faster in
  // parallel in the execution environment using the device adapter algorithms.
  std::sort(vtkm::cont::ArrayPortalToIteratorBegin(readwritePortal),
            vtkm::cont::ArrayPortalToIteratorEnd(readwritePortal));

  PortalConstType readPortal = arrayHandle.GetPortalConstControl();
  for (vtkm::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
  {
    if (readPortal.Get(index-1) > readPortal.Get(index))
    {
      std::cout << "Sorting is wrong!" << std::endl;
      break;
    }
  }
}
```

6.2.3 Interface to Execution Environment

One of the main functions of the array handle is to allow an array to be defined in the control environment and then be used in the execution environment. When using an ArrayHandle with worklets [OR FILTERS?], this transition is handled automatically. However, it is also possible to invoke the transfer for use in your own scheduled algorithms.

The ArrayHandle class manages the transition from control to execution with a set of three methods that allocate, transfer, and ready the data in one operation. These methods all start with the prefix Prepare and are meant to be called before some operation happens in the execution environment. The methods are as follows.

PrepareForInput Copies data from the control to the execution environment, if necessary, and readies the data for read-only access.

PrepareForInPlace Copies the data from the control to the execution environment, if necessary, and readies the data for both reading and writing.

PrepareForOutput Allocates space (the size of which is given as a parameter) in the execution environment, if necessary, and readies the space for writing.

The PrepareForInput and PrepareForInPlace methods each take a single argument that is the device adapter tag where execution will take place (see Section 6.1 for more information on device adapter tags). PrepareForOutput takes two arguments: the size of the space to allocate and the device adapter tag. Each of these methods returns an array portal that can be used in the execution environment. PrepareForInput returns an object of type ArrayHandle::ExecutionTypes<*DeviceAdapterTag*>::PortalConst whereas PrepareForInPlace and PrepareForOutput each return an object of type ArrayHandle::ExecutionTypes<*DeviceAdapterTag*>::Portal.

Although these Prepare methods are called in the control environment, the returned array portal can only be used in the execution environment. Thus, the portal must be passed to an invocation of the execution environment. Typically this is done with a call to Schedule in vtkm::cont::DeviceAdapterAlgorithm. This and other device adapter algorithms are described in detail in Section 6.7, but here is a quick example of using these execution array portals in a simple functor.

Example 6.11: Using an execution array portal from an ArrayHandle.

```
template<typename T, typename Device>
struct DoubleFunctor : public vtkm::exec::FunctorBase
{
  typedef typename vtkm::cont::ArrayHandle<T>::
      template ExecutionTypes<Device>::PortalConst InputPortalType;
  typedef typename vtkm::cont::ArrayHandle<T>::
      template ExecutionTypes<Device>::Portal OutputPortalType;

  VTKM_CONT_EXPORT
  DoubleFunctor(InputPortalType inputPortal, OutputPortalType outputPortal)
```

```
    : InputPortal(inputPortal), OutputPortal(outputPortal) {  }

  VTKM_EXEC_EXPORT
  void operator()(vtkm::Id index) const {
    this->OutputPortal.Set(index, 2*this->InputPortal.Get(index));
  }

  InputPortalType InputPortal;
  OutputPortalType OutputPortal;
};

template<typename T, typename Device>
void DoubleArray(vtkm::cont::ArrayHandle<T> inputArray,
                 vtkm::cont::ArrayHandle<T> outputArray,
                 Device)
{
  vtkm::Id numValues = inputArray.GetNumberOfValues();

  DoubleFunctor<T, Device> functor(
        inputArray.PrepareForInput(Device()),
        outputArray.PrepareForOutput(numValues, Device()));

  vtkm::cont::DeviceAdapterAlgorithm<Device>::Schedule(functor, numValues);
}
```

It should be noted that the array handle will expect any use of the execution array portal to finish before the next call to any `ArrayHandle` method. Since these `Prepare` methods are typically used in the process of scheduling an algorithm in the execution environment, this is seldom an issue.

6.2.4 Basic Storage

As previously discussed, `vtkm::cont::ArrayHandle` takes two template arguments.

Example 6.12: Declaration of the `vtkm::cont::ArrayHandle` templated class (again).

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
class ArrayHandle;
```

The first argument is the only one required and has been demonstrated multiple times before. The second (optional) argument specifies something called a storage, which provides the interface between the generic `vtkm::cont::ArrayHandle` class and a specific storage mechanism in the control environment.

In this and the following sections we describe this storage mechanism. A default storage is specified in much the same way as a default device adapter is defined. It is done by setting the `VTKM_STORAGE` macro. This macro must be set before including any VTK-m header files. Currently the only practical storage provided by VTK-m is the basic storage, which simply allocates a continuous section of memory of the given base type. This storage can be explicitly specified by setting `VTKM_STORAGE` to `VTKM_STORAGE_BASIC` although the basic storage will also be used as the default if no other storage is specified (which is typical).

The default storage can always be overridden by specifying an array storage tag. The tag for the basic storage is located in the vtkm/cont/StorageBasic.h header file and is named vtkm::cont::StorageTagBasic. Here is an example of specifying the storage type when declaring an array handle.

Example 6.13: Specifying the storage type for an ArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32,vtkm::cont::StorageTagBasic> arrayHandle;
```

VTK-m also defines a macro named VTKM_DEFAULT_STORAGE_TAG that can be used in place of an explicit storage tag to use the default tag. This macro is used to create new templates that have template parameters for storage that can use the default.

6.2.5 Adapting Data Structures

The intention of the storage parameter for vtkm::cont::ArrayHandle is to implement the strategy design pattern to enable VTK-m to interface directly with the data of any third party code source. VTK-m is designed to work with data originating in other libraries or applications. By creating a new type of storage, VTK-m can be entirely adapted to new kinds of data structures.

In this section we demonstrate the steps required to adapt the array handle to a data structure provided by a third party. For the purposes of the example, let us say that some fictitious library named “foo” has a simple structure named FooFields that holds the field values for a particular part of a mesh, and then maintain the field values for all locations in a mesh in a std::deque object.

Example 6.14: Fictitious field storage used in custom array storage examples.

```
#include <deque>

struct FooFields {
  float Pressure;
  float Temperature;
  float Velocity[3];
  // And so on...
};

typedef std::deque<FooFields> FooFieldsDeque;
```

VTK-m expects separate arrays for each of the fields rather than a single array containing a structure holding all of the fields. However, rather than copy each field to its own array, we can create a storage for each field that points directly to the data in a FooFieldsDeque object.

The first step in creating an adapter storage is to create a control environment array portal to the data. This is described in more detail in Section 6.2.2 and is generally straightforward for simple containers like this. Here is an example implementation for our FooFieldsDeque container.

Example 6.15: Array portal to adapt a third-party container to VTK-m.

```
#include <vtkm/cont/Assert.h>
#include <vtkm/cont/internal/IteratorFromArrayPortal.h>

// DequeType expected to be either FooFieldsDeque or const FooFieldsDeque
template<typename DequeType>
class ArrayPortalFooPressure
{
public:
  typedef float ValueType;

  VTKM_CONT_EXPORT
  ArrayPortalFooPressure() : Container(NULL) {  }

  VTKM_CONT_EXPORT
  ArrayPortalFooPressure(DequeType *container) : Container(container) {  }

  // Required to copy compatible types of ArrayPortalFooPressure. Really needed
  // to copy from non-const to const versions of array portals.
  template<typename OtherDequeType>
  VTKM_CONT_EXPORT
  ArrayPortalFooPressure(const ArrayPortalFooPressure<OtherDequeType> &other)
    : Container(other.GetContainer()) {  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT_EXPORT
  ValueType Get(vtkm::Id index) const {
    VTKM_ASSERT_CONT(index >= 0);
    VTKM_ASSERT_CONT(index < this->GetNumberOfValues());
    return (*this->Container)[index].Pressure;
  }

  VTKM_CONT_EXPORT
  void Set(vtkm::Id index, ValueType value) const {
    VTKM_ASSERT_CONT(index >= 0);
    VTKM_ASSERT_CONT(index < this->GetNumberOfValues());
    (*this->Container)[index].Pressure = value;
  }

  // Here for the copy constructor.
  VTKM_CONT_EXPORT
  DequeType *GetContainer() const { return this->Container; }

private:
  DequeType *Container;
};
```

The next step in creating an adapter storage is to define a tag for the adapter. We shall call ours `StorageTagFooPressure`. Then, we need to create a specialization of the templated `vtkm::cont::internal::Storage` class. The `ArrayHandle` will instantiate an object using the array container tag we give it, and we define our own specialization so that it runs our interface into the code.

`vtkm::cont::internal::Storage` has two template arguments: the base type of the array and the storage tag.

Example 6.16: Prototype for `vtkm::cont::internal::Storage`.

```
namespace vtkm {
```

```
namespace cont {
namespace internal {

template<typename T, class StorageTag>
class Storage;

}
}
} // namespace vtkm::cont::internal
```

The `vtkm::cont::internal::Storage` must define the following items.

`ValueType` A `typedef` of the type for each item in the array. This is the same type as the first template argument.

`PortalType` The type of an array portal that can be used to access the underlying data. This array portal needs to work only in the control environment.

`PortalConstType` A read-only (const) version of `PortalType`.

`GetPortal` A method that returns an array portal of type `PortalType` that can be used to access the data manged in this storage.

`GetPortalConst` Same as `GetPortal` except it returns a read-only (const) array portal.

`GetNumberOfValues` A method that returns the number of values the storage is currently allocated for.

`Allocate` A method that allocates the array to a given size. An values stored in the previous allocation may be destroyed.

`Shrink` A method like `Allocate` with two differences. First, the size of the allocation must be smaller than the existing allocation when the method is called. Second, any values currently stored in the array will be valid after the array is resized. This constrained form of allocation allows the array to be resized and values valid without ever having to copy data.

`ReleaseResources` A method that instructs the storage to free all of its memory.

The following provides an example implementation of our adapter to a `FooFieldsDeque`. It relies on the `ArrayPortalFooPressure` provided in Example 6.15.

Example 6.17: Storage to adapt a third-party container to VTK-m.

```
// Includes or definition for ArrayPortalFooPressure

struct StorageTagFooPressure {  };

namespace vtkm {
namespace cont {
namespace internal {

template<>
class Storage<float, StorageTagFooPressure>
{
public:
  typedef float ValueType;

  typedef ArrayPortalFooPressure<FooFieldsDeque> PortalType;
  typedef ArrayPortalFooPressure<const FooFieldsDeque> PortalConstType;

  VTKM_CONT_EXPORT
  Storage() : Container(NULL) {  }

  VTKM_CONT_EXPORT
  Storage(FooFieldsDeque *container) : Container(container) {  }

  VTKM_CONT_EXPORT
  PortalType GetPortal() { return PortalType(this->Container); }

  VTKM_CONT_EXPORT
  PortalConstType GetPortalConst() const {
    return PortalConstType(this->Container);
  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT_EXPORT
  void Allocate(vtkm::Id numberOfValues) {
    this->Container->resize(numberOfValues);
  }

  VTKM_CONT_EXPORT
  void Shrink(vtkm::Id numberOfValues) {
    this->Container->resize(numberOfValues);
  }

  VTKM_CONT_EXPORT
  void ReleaseResources() { this->Container->clear(); }

private:
  FooFieldsDeque *Container;
};

}
}
} // namespace vtkm::cont::internal
```

The final step to make a storage adapter is to make a mechanism to construct an `ArrayHandle` that points to a particular storage. This can be done by creating a trivial subclass of `vtkm::cont::ArrayHandle` that simply constructs the array handle to the state of an existing container.

Example 6.18: Array handle to adapt a third-party container to VTK-m.

```
class ArrayHandleFooPressure
    : public vtkm::cont::ArrayHandle<float, StorageTagFooPressure>
{
private:
  typedef vtkm::cont::internal::Storage<float, StorageTagFooPressure>
      StorageType;

public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
      ArrayHandleFooPressure,
      (vtkm::cont::ArrayHandle<float, StorageTagFooPressure>));
  VTKM_CONT_EXPORT
  ArrayHandleFooPressure(FooFieldsDeque *container)
    : Superclass(StorageType(container)) {  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the typedefs Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_-HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 6.18 is not templated, so it uses the VTKM_ARRAY_-HANDLE_SUBCLASS_NT macro. (The other macro is described in Section 6.2.7 on page 59). This macro takes two parameters. The first parameter is the name of the subclass where the macro is defined and the second parameter is the immediate superclass including the full template specification. The second parameter of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

With this new version of ArrayHandle, VTK-m can now read to and write from the FooFieldsDeque structure directly. Note, however, that when writing to an array handle, it is necessary to call GetPortalControl or GetPortalConstControl to flush data from the execution environment to the control environment. [Should probably make this easier.]

[Although this example was originally written with executing worklets in mind, at this level of the user's guide it would probably be better to have an example that uses a filter. Once the filters are settled, that is.]

Example 6.19: Using an ArrayHandle with custom container.

```
template<typename GridType>
VTKM_CONT_EXPORT
void GetElevationAirPressure(const GridType &grid, FooFieldsDeque *fields)
{
  // Make an array handle that points to the pressure values in the fields.
  ArrayHandleFooPressure pressureHandle(fields);

  // This is currently commented out because worklets are not yet implemented.
//  // Run an elevation worklet.
//  vtkm::worklet::Elevation elevation(vtkm::make_Vector3(0.0, 0.0, 0.0),
//                                     vtkm::make_Vector3(0.0, 0.0, 10.0),
//                                     vtkm::make_Vector2(0.02, 0.0));
//  vtkm::cont::DispatcherMapField<vtkm::worklet::Elevation>
//      dispatcher(elevation);
//  dispatcher.Invoke(grid.GetPointCoordinates(), pressureHandle);

  // Make sure the values are flushed back to the control environment.
  pressureHandle.GetPortalConstControl();

  // Now the pressure field is in the fields container.
}
```

6.2.6 Implicit Array Handles

The generic array handle and storage templating in VTK-m allows for any type of operations to retrieve a particular value. Typically this is used to convert an index to some location or locations in memory. However, it is also possible to compute a value directly from an index rather than look up some value in memory. Such an array is completely functional and requires no storage in memory at all. Such a functional array is called an *implicit array handle*. Implicit arrays are an example of *fancy array handles*, which are array handles that behave like regular arrays but do special processing under the covers to provide values.

Specifying a functional or implicit array in VTK-m is straightforward. VTK-m has a special class named `vtkm::cont::ArrayHandleImplicit` that makes an implicit array containing values generated by a user-specified *functor*. A functor is simply a C++ class or struct that contains an overloaded parenthesis operator so that it can be used syntactically like a function.

To demonstrate the use of `ArrayHandleImplicit`, let us say we want an array of even numbers. The array has the values $[0, 2, 4, 6, \ldots]$ (double the index) up to some given size. Although we could easily create this array in memory, we can save space and possibly time by computing these values on demand.

The first step to using `ArrayHandleImplicit` is to declare a functor. The functor's parenthesis operator should accept a single argument of type `vtkm::Id` and return a value appropriate for that index. The parenthesis operator should also be declared `const` because it is not allowed to change the class' state.

Example 6.20: Functor that doubles an index.

```
struct DoubleIndexFunctor
{
  VTKM_EXEC_CONT_EXPORT
  vtkm::Id operator()(vtkm::Id index) const
  {
    return 2*index;
  }
};
```

Once the functor is defined, an implicit array can be declared using the templated `vtkm::cont::ArrayHandleImplicit` class. The first template argument is the type of the array's values (which should match the return value for the functor), and the second template argument is the functor type.

Example 6.21: Declaring a `ArrayHandleImplicit`.

```
  vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
      implicitArray(DoubleIndexFunctor(), 50);
```

For convenience, vtkm/cont/ArrayHandleImplicit.h also declares the `vtkm::cont::make_ArrayHandleImplicit` function. This function takes a functor and the size of the array and returns the implicit array. When using this function, you also have to declare the first template argument, which is the array's value type, since this type does not appear in any of the arguments.

Example 6.22: Using make_ArrayHandleImplicit.

```
  vtkm::cont::make_ArrayHandleImplicit<vtkm::Id>(DoubleIndexFunctor(), 50);
```

If the implicit array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of vtkm::cont::-ArrayHandleImplicit for your array.

Example 6.23: Custom implicit array handle for even numbers.

```
#include <vtkm/cont/ArrayHandleImplicit.h>

class ArrayHandleDoubleIndex
    : public vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
      ArrayHandleDoubleIndex,
      (vtkm::cont::ArrayHandleImplicit<vtkm::Id,DoubleIndexFunctor>));

  VTKM_CONT_EXPORT
  ArrayHandleDoubleIndex(vtkm::Id numberOfValues)
    : Superclass(DoubleIndexFunctor(), numberOfValues) {  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the typedefs Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_-HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 6.23 is not templated, so it uses the VTKM_ARRAY_-HANDLE_SUBCLASS_NT macro. (The other macro is described in Section 6.2.7 on page 59). This macro takes two parameters. The first parameter is the name of the subclass where the macro is defined and the second parameter is the immediate superclass including the full template specification. The second parameter of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

VTK-m comes with some examples of implicit storage. vtkm::cont::ArrayHandleCon-stant returns the same value for every index in the array. The constant array is useful when an algorithm that can work on a variable field is used on a constant value. vtkm::cont::-ArrayHandleIndex returns the index as the value. vtkm::cont::ArrayHandleCounting is an array that starts from a given value and then counts with a given step. The index and counting arrays are useful for generating fields of identifiers or for indexing operations. vtkm::cont::ArrayHandleUniformPointCoordinates generates the position of points in axis-aligned uniform rectilinear grids. It is used internally by VTK-m to create point coordinates for these types of data structures.

6.2.7 Transformed Arrays

Another type of fancy array handle is the transformed array. A transformed array takes another array and applies a function to all of the elements to produce a new array. A transformed array behaves much like a map operation except that a map operation writes its values to a new memory location whereas the transformed array handle produces its values on demand so that no additional storage is required.

Specifying a transformed array in VTK-m is straightforward. VTK-m has a special class named `vtkm::cont::ArrayHandleTransform` that takes an array handle and a functor and provides an interface to a new array comprising values of the first array applied to the functor.

To demonstrate the use of `ArrayHandleTransform`, let us say that we want to scale and bias all of the values in a target array. That is, each value in the target array is going to be multiplied by a given scale and then offset by adding a bias value. (The scale and bias are uniform across all entries.) We could, of course, easily create a worklet to apply this scale and bias to each entry in the target array and save the result in a new array, but we can save space and possibly time by computing these values on demand.

The first step to using `ArrayHandleTransform` is to declare a functor. The functor's parenthesis operator should accept a single argument of the type of the target array and return the transformed value. For more generally applicable transform functors, it is often useful to make the parenthesis operator a template. The parenthesis operator should also be declared `const` because it is not allowed to change the class' state.

Example 6.24: Functor to scale and bias a value.

```
template<typename T>
struct ScaleBiasFunctor
{
  VTKM_EXEC_CONT_EXPORT
  ScaleBiasFunctor(T scale = T(1), T bias = T(0))
    : Scale(scale), Bias(bias) {  }

  VTKM_EXEC_CONT_EXPORT
  T operator()(T x) const
  {
    return this->Scale*x + this->Bias;
  }

  T Scale;
  T Bias;
};
```

Once the functor is defined, a transformed array can be declared using the templated `vtkm::cont::ArrayHandleTransform` class. The first template argument is the type of the array's values (which should match the return value for the functor). The second template argument is the type of array being transformed. The third and final template argument is the type of functor used for the transformation.

That said, it is generally easier to use the `vtkm::cont::make_ArrayHandleTransform` convenience function. This function takes an array and a functor and returns a transformed array. When using this function, you also have to declare the first template argument, which is the transformed array's value type, since this type does not appear in any of the arguments.

Example 6.25: Using `make_ArrayHandleTransform`.

```
  vtkm::cont::make_ArrayHandleTransform<vtkm::Float32>(
          array, ScaleBiasFunctor<vtkm::Float32>(2,3))
```

If the transformed array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of `vtkm::cont::ArrayHandleTransform` or convenience `make_ArrayHandle*` function for your array.

Example 6.26: Custom transform array handle for scale and bias.

```
#include <vtkm/cont/ArrayHandleTransform.h>

template<typename ArrayHandleType>
class ArrayHandleScaleBias
    : public vtkm::cont::ArrayHandleTransform<
          typename ArrayHandleType::ValueType,
          ArrayHandleType,
          ScaleBiasFunctor<typename ArrayHandleType::ValueType> >
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
      ArrayHandleScaleBias,
      (ArrayHandleScaleBias<ArrayHandleType>),
      (vtkm::cont::ArrayHandleTransform<
         typename ArrayHandleType::ValueType,
         ArrayHandleType,
         ScaleBiasFunctor<typename ArrayHandleType::ValueType> >)
      );

  VTKM_CONT_EXPORT
  ArrayHandleScaleBias(const ArrayHandleType &array,
                       ValueType scale,
                       ValueType bias)
    : Superclass(array, ScaleBiasFunctor<ValueType>(scale, bias)) {  }
};

template<typename ArrayHandleType>
VTKM_CONT_EXPORT
ArrayHandleScaleBias<ArrayHandleType>
make_ArrayHandleScaleBias(const ArrayHandleType &array,
                          typename ArrayHandleType::ValueType scale,
                          typename ArrayHandleType::ValueType bias)
{
  return ArrayHandleScaleBias<ArrayHandleType>(array, scale, bias);
}
```

Subclasses of `ArrayHandle` provide constructors that establish the state of the array handle. All array handle subclasses must also use either the `VTKM_ARRAY_HANDLE_SUBCLASS` macro or the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. Both of these macros define the typedefs `Superclass`, `ValueType`, and `StorageTag` as well as a set of constructors and operators expected of all `ArrayHandle` classes. The difference between these two macros is that `VTKM_ARRAY_HANDLE_SUBCLASS` is used in templated classes whereas `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is used in non-templated classes.

The `ArrayHandle` subclass in Example 6.26 is templated, so it uses the `VTKM_ARRAY_-`

HANDLE_SUBCLASS macro. (The other macro is described in Section 6.2.5 on page 55). This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

6.2.8 Permuted Arrays

A permutation array is a fancy array handle that reorders the elements in an array. Elements in the array can be skipped over or replicated. The permutation array provides this reordered array without actually coping any data. Instead, indices are adjusted as the array is accessed.

Specifying a permutation array in VTK-m is straightforward. VTK-m has a class named `vtkm::cont::ArrayHandlePermutation` that takes two arrays: an array of values and an array of indices that maps an index in the permutation to an index of the original values. The index array is specified first. The following example is a simple demonstration of the permutation array handle.

Example 6.27: Using `ArrayHandlePermutation`.

```
typedef vtkm::cont::ArrayHandle<vtkm::Id> IdArrayType;
typedef IdArrayType::PortalControl IdPortalType;

typedef vtkm::cont::ArrayHandle<vtkm::Float32> ValueArrayType;
typedef ValueArrayType::PortalControl ValuePortalType;

// Create array with values [0.0, 0.1, 0.2, 0.3]
ValueArrayType valueArray;
valueArray.Allocate(4);
ValuePortalType valuePortal = valueArray.GetPortalControl();
valuePortal.Set(0, 0.0);
valuePortal.Set(1, 0.1);
valuePortal.Set(2, 0.2);
valuePortal.Set(3, 0.3);

// Use ArrayHandlePermutation to make an array = [0.3, 0.0, 0.1].
IdArrayType idArray1;
idArray1.Allocate(3);
IdPortalType idPortal1 = idArray1.GetPortalControl();
idPortal1.Set(0, 3);
idPortal1.Set(1, 0);
idPortal1.Set(2, 1);
vtkm::cont::ArrayHandlePermutation<IdArrayType,ValueArrayType>
    permutedArray1(idArray1, valueArray);

// Use ArrayHandlePermutation to make an array = [0.1, 0.2, 0.2, 0.3, 0.0]
IdArrayType idArray2;
idArray2.Allocate(5);
IdPortalType idPortal2 = idArray2.GetPortalControl();
idPortal2.Set(0, 1);
idPortal2.Set(1, 2);
idPortal2.Set(2, 2);
idPortal2.Set(3, 3);
idPortal2.Set(4, 0);
vtkm::cont::ArrayHandlePermutation<IdArrayType,ValueArrayType>
    permutedArray2(idArray2, valueArray);
```

The vtkm/cont/ArrayHandlePermutation.h header also contains the templated convenience function vtkm::cont::make_ArrayHandlePermutation that takes instances of the index and value array handles and returns a permutation array. This function can sometimes be used to avoid having to declare the full array type.

Example 6.28: Using make_ArrayHandlePermutation.

```
vtkm::cont::make_ArrayHandlePermutation(idArray,valueArray)
```

When using an ArrayHandlePermutation, take care that all the provided indices in the index array point to valid locations in the values array. Bad indices can cause reading from or writing to invalid memory locations, which can be difficult to debug.

6.2.9 Zipped Arrays

A zip array is a fancy array handle that combines two arrays of the same size to pair up the corresponding values. Each element in the zipped array is a vtkm::Pair containing the values of the two respective arrays. These pairs are not stored in their own memory space. Rather, the pairs are generated as the array is used.

Specifying a zipped array in VTK-m is straightforward. VTK-m has a class named vtkm::cont::ArrayHandleZip that takes the two arrays providing values for the first and second entries in the pairs. The following example is a simple demonstration of creating a zip array handle.

Example 6.29: Using ArrayHandleZip.

```
typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType1;
typedef ArrayType1::PortalControl PortalType1;

typedef vtkm::cont::ArrayHandle<vtkm::Float32> ArrayType2;
typedef ArrayType2::PortalControl PortalType2;

// Create an array of vtkm::Id with values [3, 0, 1]
ArrayType1 array1;
array1.Allocate(3);
PortalType1 portal1 = array1.GetPortalControl();
portal1.Set(0, 3);
portal1.Set(1, 0);
portal1.Set(2, 1);

// Create a second array of vtkm::Float32 with values [0.0, 0.1, 0.2]
ArrayType2 array2;
array2.Allocate(3);
PortalType2 portal2 = array2.GetPortalControl();
portal2.Set(0, 0.0);
portal2.Set(1, 0.1);
portal2.Set(2, 0.2);

// Zip the two arrays together to create an array of
// vtkm::Pair<vtkm::Id, vtkm::Float32> with values [(3,0.0), (0,0.1), (1,0.2)]
vtkm::cont::ArrayHandleZip<ArrayType1,ArrayType2> zipArray(array1, array2);
```

The vtkm/cont/ArrayHandleZip.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleZip that takes instances of the two array handles and returns a zip array. This function can sometimes be used to avoid having to declare the full array type.

Example 6.30: Using make_ArrayHandleZip.

```
vtkm::cont::make_ArrayHandleZip(array1,array2)
```

6.2.10 Derived Storage

So far, we have discussed using the array storage mechanism to adapt to particular memory layout and to create implicit and other special fancy arrays. Yet another option is to create a *derived storage*. A derived storage allows you to declare your own type of fancy array. A derived storage takes one or more other arrays and changes their behavior in some way. Their implementation is similar to adapting a memory layout, but some of the details are different.

In this section we will demonstrate the steps required to create a derived storage. A transform, permutation, and zip arrays, documented in Sections 6.2.7, 6.2.8, and 6.2.9, respectively, provide specific forms of a derived storage. When applicable, it is much easier to create a derived array using these pre-implemented fancy arrays than to create your own derived storage. However, if these pre-existing fancy arrays do not work work, for example if your derivation uses multiple arrays or requires general lookups, you can do so by creating your own derived storage. For the purposes of the example in this section, let us say we want 2 array handles to behave as one array with the contents concatenated together. We could of course actually copy the data, but we can also do it in place.

The first step to creating a derived storage is to build an array portal that will take portals from arrays being derived. The portal must work in both the control and execution environment (or have a separate version for control and execution).

Example 6.31: Derived array portal for concatenated arrays.

```
#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/ArrayPortal.h>
#include <vtkm/cont/Assert.h>

template<typename P1, typename P2>
class ArrayPortalConcatenate
{
public:
  typedef P1 PortalType1;
  typedef P2 PortalType2;
  typedef typename PortalType1::ValueType ValueType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT_EXPORT
  ArrayPortalConcatenate() : Portal1(), Portal2() {  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT_EXPORT
  ArrayPortalConcatenate(const PortalType1 &portal1, const PortalType2 portal2)
    : Portal1(portal1), Portal2(portal2) {  }

  /// Copy constructor for any other ArrayPortalConcatenate with a portal type
  /// that can be copied to this portal type. This allows us to do any type
  /// casting that the portals do (like the non-const to const cast).
  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OtherP1, typename OtherP2>
  VTKM_EXEC_CONT_EXPORT
  ArrayPortalConcatenate(const ArrayPortalConcatenate<OtherP1,OtherP2> &src)
    : Portal1(src.GetPortal1()), Portal2(src.GetPortal2()) {  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return
        this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT_EXPORT
  ValueType Get(vtkm::Id index) const {
    if (index < this->Portal1.GetNumberOfValues())
    {
      return this->Portal1.Get(index);
    }
    else
    {
      return this->Portal2.Get(index - this->Portal1.GetNumberOfValues());
    }
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT_EXPORT
  void Set(vtkm::Id index, const ValueType &value) const {
    if (index < this->Portal1.GetNumberOfValues())
    {
      this->Portal1.Set(index, value);
    }
    else
    {
      this->Portal2.Set(index - this->Portal1.GetNumberOfValues(), value);
    }
  }

  VTKM_EXEC_CONT_EXPORT
  const PortalType1 &GetPortal1() const { return this->Portal1; }
  VTKM_EXEC_CONT_EXPORT
  const PortalType2 &GetPortal2() const { return this->Portal2; }

private:
  PortalType1 Portal1;
  PortalType2 Portal2;
};
```

Like in an adapter storage, the next step in creating a derived storage is to define a tag for the adapter. We shall call ours `StorageTagConcatenate` and it will be templated on the two array handle types that we are deriving. Then, we need to create a specialization of the templated `vtkm::cont::internal::Storage` class. The implementation for a `Storage` for a derived storage is usually trivial compared to an adapter storage because the majority of the work is deferred to the derived arrays.

Example 6.32: `Storage` for derived container of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
struct StorageTagConcatenate {  };

namespace vtkm {
```

```
namespace cont {
namespace internal {

template<typename ArrayHandleType1, typename ArrayHandleType2>
class Storage<
    typename ArrayHandleType1::ValueType,
    StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2> >
{
public:
  typedef typename ArrayHandleType1::ValueType ValueType;

  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::PortalControl,
      typename ArrayHandleType2::PortalControl> PortalType;
  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::PortalConstControl,
      typename ArrayHandleType2::PortalConstControl> PortalConstType;

  VTKM_CONT_EXPORT
  Storage() : Valid(false) {  }

  VTKM_CONT_EXPORT
  Storage(const ArrayHandleType1 array1, const ArrayHandleType2 array2)
    : Array1(array1), Array2(array2), Valid(true) {  }

  VTKM_CONT_EXPORT
  PortalType GetPortal() {
    VTKM_ASSERT_CONT(this->Valid);
    return PortalType(this->Array1.GetPortalControl(),
                      this->Array2.GetPortalControl());
  }

  VTKM_CONT_EXPORT
  PortalConstType GetPortalConst() const {
    VTKM_ASSERT_CONT(this->Valid);
    return PortalConstType(this->Array1.GetPortalConstControl(),
                           this->Array2.GetPortalConstControl());
  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    VTKM_ASSERT_CONT(this->Valid);
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  VTKM_CONT_EXPORT
  void Allocate(vtkm::Id numberOfValues) {
    VTKM_ASSERT_CONT(this->Valid);
    // This implementation of allocate, which allocates the same amount in both
    // arrays, is arbitrary. It could, for example, leave the size of Array1
    // alone and change the size of Array2. Or, probably most likely, it could
    // simply throw an error and state that this operation is invalid.
    vtkm::Id half = numberOfValues/2;
    this->Array1.Allocate(numberOfValues-half);
    this->Array2.Allocate(half);
  }

  VTKM_CONT_EXPORT
  void Shrink(vtkm::Id numberOfValues) {
    VTKM_ASSERT_CONT(this->Valid);
    if (numberOfValues < this->Array1.GetNumberOfValues())
    {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
    }
    else
    {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
    }
  }

  VTKM_CONT_EXPORT
  void ReleaseResources() {
    VTKM_ASSERT_CONT(this->Valid);
    this->Array1.ReleaseResources();
    this->Array2.ReleaseResources();
  }

  // Requried for later use in ArrayTransfer class.
  VTKM_CONT_EXPORT
  const ArrayHandleType1 &GetArray1() const {
    VTKM_ASSERT_CONT(this->Valid);
    return this->Array1;
  }
  VTKM_CONT_EXPORT
  const ArrayHandleType2 &GetArray2() const {
    VTKM_ASSERT_CONT(this->Valid);
    return this->Array2;
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
  bool Valid;
};

}
}
} // namespace vtkm::cont::internal
```

One of the responsibilities of an array handle is to copy data between the control and execution environments. The default behavior is to request the device adapter to copy data items from one environment to another. This might involve transferring data between a host and device. For an array of data resting in memory, this is necessary. However, implicit storage (described in the previous section) overrides this behavior to pass nothing but the functional array portal. Likewise, it is undesirable to do a raw transfer of data with derived storage. The underlying arrays being derived may be used in other contexts, and it would be good to share the data wherever possible. It is also sometimes more efficient to copy data independently from the arrays being derived than from the derived storage itself.

The mechanism that controls how a particular storage gets transferred to and from the execution environment is encapsulated in the templated `vtkm::cont::internal::ArrayTransfer` class. By creating a specialization of `vtkm::cont::internal::ArrayTransfer`, we can modify the transfer behavior to instead transfer the arrays being derived and use the respective copies in the control and execution environments.

`vtkm::cont::internal::ArrayTransfer` has three template arguments: the base type of the array, the storage tag, and the device adapter tag.

Example 6.33: Prototype for `vtkm::cont::internal::ArrayTransfer`.

```
namespace vtkm {
namespace cont {
namespace internal {

template<typename T,typename StorageTag,typename DeviceAdapterTag>
class ArrayTransfer;
}
}
} //namespace vtkm::cont::internal
```

All vtkm::cont::internal::ArrayTransfer implementations must have a constructor method that accepts a pointer to a vtkm::cont::internal::Storage object templated to the same base type and storage tag as the ArrayTransfer object. Assuming that an ArrayHandle is templated using the parameters in Example 6.33, the prototype for the constructor must be equivalent to the following.

Example 6.34: Prototype for ArrayTransfer constructor.

```
ArrayTransfer(vtkm::cont::internal::Storage<T, StorageTag> *storage);
```

Typically the constructor either saves the Storage pointer or other relevant objects from the Storage for later use in the methods.

In addition to this non-default constructor, the vtkm::cont::internal::ArrayTransfer specialization must define the following items.

ValueType A typedef of the type for each item in the array. This is the same type as the first template argument.

PortalControl The type of an array portal that is used to access the underlying data in the control environment.

PortalConstControl A read-only (const) version of PortalControl.

PortalExecution The type of an array portal that is used to access the underlying data in the execution environment.

PortalConstExecution A read-only (const) version of PortalExecution.

GetNumberOfValues A method that returns the number of values currently allocated in the execution environment. The results may be undefined if none of the load or allocate methods have yet been called.

PrepareForInput A method responsible for transferring data from the control to the execution for input. PrepareForInput has one Boolean argument that controls whether this transfer should actually take place. When true, data from the Storage object given in the constructor should be transferred to the execution environment; otherwise the data should not be copied. An ArrayTransfer for a derived array typically ignores this parameter since the arrays being derived manages this transfer already. Regardless of the Boolean flag, a PortalConstExecution is returned.

PrepareForInPlace A method that behaves just like PrepareForInput except that the data in the execution environment is used for both reading and writing so the method returns a PortalExecution. If the array is considered read-only, which is common for derived arrays, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`PrepareForOutput` A method that takes a size (in a `vtkm::Id`) and allocates an array in the execution environment of the specified size. The initial memory can be uninitialized. The method returns a `PortalExecution` for the allocated data. If the array is considered read-only, which is common for derived arrays, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`RetrieveOutputData` This method takes an array storage pointer (which is the same as that passed to the constructor, but provided for convenience), allocates memory in the control environment, and copies data from the execution environment into it. If the derived array is considered read-only and both `PrepareForInPlace` and `PrepareForOutput` throw exceptions, then this method should never be called. If it is, then that is probably a bug in `ArrayHandle`, and it is OK to throw `vtkm::cont::ErrorControlInternal`.

`Shrink` A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked. If the derived array is considered read-only, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`ReleaseResources` A method that frees any resources (typically memory) in the execution environment.

Continuing our example derived storage that concatenates two arrays started in Examples 6.31 and 6.32, the following provides an `ArrayTransfer` appropriate for the derived storage.

Example 6.35: `ArrayTransfer` for derived storage of concatenated arrays.

```
namespace vtkm {
namespace cont {
namespace internal {

template<typename ArrayHandleType1,
         typename ArrayHandleType2,
         typename Device>
class ArrayTransfer<
    typename ArrayHandleType1::ValueType,
    StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2>,
    Device>
{
public:
  typedef typename ArrayHandleType1::ValueType ValueType;

private:
  typedef StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2>
      StorageTag;
  typedef vtkm::cont::internal::Storage<ValueType,StorageTag>
      StorageType;

public:
  typedef typename StorageType::PortalType PortalControl;
  typedef typename StorageType::PortalConstType PortalConstControl;
  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::template ExecutionTypes<Device>::Portal,
      typename ArrayHandleType2::template ExecutionTypes<Device>::Portal>
    PortalExecution;
  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::template ExecutionTypes<Device>::PortalConst,
      typename ArrayHandleType2::template ExecutionTypes<Device>::PortalConst>
    PortalConstExecution;

  VTKM_CONT_EXPORT
  ArrayTransfer(StorageType *storage)
    : Array1(storage->GetArray1()), Array2(storage->GetArray2())
  { }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  VTKM_CONT_EXPORT
  PortalConstExecution PrepareForInput(bool vtkmNotUsed(updateData)) {
    return PortalConstExecution(this->Array1.PrepareForInput(Device()),
                                this->Array2.PrepareForInput(Device()));
  }

  VTKM_CONT_EXPORT
  PortalExecution PrepareForInPlace(bool vtkmNotUsed(updateData)) {
    return PortalExecution(this->Array1.PrepareForInPlace(Device()),
                           this->Array2.PrepareForInPlace(Device()));
  }

  VTKM_CONT_EXPORT
  PortalExecution PrepareForOutput(vtkm::Id numberOfValues)
  {
    // This implementation of allocate, which allocates the same amount in both
    // arrays, is arbitrary. It could, for example, leave the size of Array1
    // alone and change the size of Array2. Or, probably most likely, it could
    // simply throw an error and state that this operation is invalid.
    vtkm::Id half = numberOfValues/2;
    return PortalExecution(
          this->Array1.PrepareForOutput(numberOfValues-half, Device()),
          this->Array2.PrepareForOutput(half, Device()));
  }

  VTKM_CONT_EXPORT
  void RetrieveOutputData(StorageType *vtkmNotUsed(storage)) const {
    // Implementation of this method should be unnecessary. The internal
    // array handles should automatically retrieve the output data as
    // necessary.
  }

  VTKM_CONT_EXPORT
  void Shrink(vtkm::Id numberOfValues) {
    if (numberOfValues < this->Array1.GetNumberOfValues())
    {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
    }
    else
    {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
    }
  }

  VTKM_CONT_EXPORT
  void ReleaseResources() {
    this->Array1.ReleaseResourcesExecution();
```

```
    this->Array2.ReleaseResourcesExecution();
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
};

}
}
} // namespace vtkm::cont::internal
```

The final step to make a derived storage is to create a mechanism to construct an ArrayHandle with a storage derived from the desired arrays. This can be done by creating a trivial subclass of vtkm::cont::ArrayHandle that simply constructs the array handle to the state of an existing storage. It uses a protected constructor of vtkm::cont::ArrayHandle that accepts a constructed storage.

Example 6.36: ArrayHandle for derived storage of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
class ArrayHandleConcatenate
    : public vtkm::cont::ArrayHandle<
        typename ArrayHandleType1::ValueType,
        StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2> >
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
      ArrayHandleConcatenate,
      (ArrayHandleConcatenate<ArrayHandleType1,ArrayHandleType2>),
      (vtkm::cont::ArrayHandle<
         typename ArrayHandleType1::ValueType,
         StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2> >));

private:
  typedef vtkm::cont::internal::Storage<ValueType,StorageTag> StorageType;

public:
  VTKM_CONT_EXPORT
  ArrayHandleConcatenate(const ArrayHandleType1 &array1,
                         const ArrayHandleType2 &array2)
    : Superclass(StorageType(array1, array2)) {  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the typedefs Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 6.36 is templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS macro. (The other macro is described in Section 6.2.5 on page 55). This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

vtkm::cont::ArrayHandleCompositeVector is an example of a derived array handle provided by VTK-m. It references some fixed number of other arrays, pulls a specified component out of each, and produces a new component that is a tuple of these retrieved components.

6.3 Dynamic Array Handle

The ArrayHandle class uses templating to make very efficient and type-safe access to data. However, it is sometimes inconvenient or impossible to specify the element type and storage at run-time. The DynamicArrayHandle class provides a mechanism to manage arrays of data with unspecified types.

vtkm::cont::DynamicArrayHandle holds a reference to an array. Unlike ArrayHandle, DynamicArrayHandle is *not* templated. Instead, it uses C++ run-type type information to store the array without type and cast it when appropriate.

A DynamicArrayHandle can be established by constructing it with or assigning it to an ArrayHandle. The following example demonstrates how a DynamicArrayHandle might be used to load an array whose type is not known until run-time.

Example 6.37: Creating a DynamicArrayHandle.

```
VTKM_CONT_EXPORT
vtkm::cont::DynamicArrayHandle
LoadDynamicArray(const void *buffer, vtkm::Id length, std::string type)
{
  vtkm::cont::DynamicArrayHandle handle;
  if (type == "float")
  {
    vtkm::cont::ArrayHandle<vtkm::Float32> concreteArray =
        vtkm::cont::make_ArrayHandle(
          reinterpret_cast<const vtkm::Float32*>(buffer), length);
    handle = concreteArray;
  } else if (type == "int") {
    vtkm::cont::ArrayHandle<vtkm::Int32> concreteArray =
        vtkm::cont::make_ArrayHandle(
          reinterpret_cast<const vtkm::Int32*>(buffer), length);
    handle = concreteArray;
  }
  return handle;
}
```

6.3.1 Querying and Casting

Data pointed to by a DynamicArrayHandle is not directly accessible. However, there are a few generic queries you can make without directly knowing the data type. The GetNumberOfValues method returns the length of the array with respect to its base data type. It is also common in VTK-m to use data types, such as vtkm::Vec, with multiple components per value. The GetNumberOfComponents method returns the number of components in a vector-like type (or 1 for scalars).

Example 6.38: Non type-specific queries on DynamicArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle scalarDynamicHandle(
      vtkm::cont::make_ArrayHandle(scalarBuffer));

// This returns 10.
vtkm::Id scalarArraySize = scalarDynamicHandle.GetNumberOfValues();

// This returns 1.
vtkm::IdComponent scalarComponents =
    scalarDynamicHandle.GetNumberOfComponents();

std::vector<vtkm::Vec<vtkm::Float32,3> > vectorBuffer(20);
vtkm::cont::DynamicArrayHandle vectorDynamicHandle(
      vtkm::cont::make_ArrayHandle(vectorBuffer));

// This returns 20.
vtkm::Id vectorArraySize = vectorDynamicHandle.GetNumberOfValues();

// This returns 3.
vtkm::IdComponent vectorComponents =
    vectorDynamicHandle.GetNumberOfComponents();
```

It is also often desirable to create a new array based on the underlying type of a DynamicArrayHandle. For example, when executing a worklet [OR FILTER?] on one or more fields, it is common to generate an output field, and it is furthermore usually desirable for the output type to match the input type. To satisfy this use case, DynamicArrayHandle has a method named NewInstance that creates a new empty array with the same underlying type as the original array.

Example 6.39: Using DynamicArrayHandle::NewInstance().

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle dynamicHandle(
      vtkm::cont::make_ArrayHandle(scalarBuffer));

// This creates a new empty array of type Float32.
vtkm::cont::DynamicArrayHandle newDynamicArray = dynamicHandle.NewInstance();
```

Before the data with a DynamicArrayHandle can be accessed, the type and storage of the array must be established. This is usually done internally within VTK-m when a worklet [OR FILTER?] is invoked. However, it is also possible to query the types and cast to a concrete ArrayHandle.

You can query the component type and storage type using the IsArrayHandleType, IsSameType, and IsTypeAndStorage methods. IsArrayHandleType takes an example array handle type and returns whether the underlying array matches the given static array type. IsSameType behaves the same as IsArrayHandleType but accepts an instances of an ArrayHandle object to automatically resolve the template parameters. IsTypeAndStorage takes an example component type and an example storage type as arguments and returns whether the underlying array matches both types.

Example 6.40: Querying the component and storage types of a DynamicArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::ArrayHandle<vtkm::Float32> concreteHandle =
    vtkm::cont::make_ArrayHandle(scalarBuffer);
vtkm::cont::DynamicArrayHandle dynamicHandle(concreteHandle);

// This returns true
bool isFloat32Array = dynamicHandle.IsSameType(concreteHandle);

// This returns false
bool isIdArray =
    dynamicHandle.IsArrayHandleType<vtkm::cont::ArrayHandle<vtkm::Id> >();

// This returns true
bool isFloat32 =
    dynamicHandle.IsTypeAndStorage<vtkm::Float32,VTKM_DEFAULT_STORAGE_TAG>();

// This returns false
bool isId =
    dynamicHandle.IsTypeAndStorage<vtkm::Id,VTKM_DEFAULT_STORAGE_TAG>();

// This returns false
bool isErrorStorage = dynamicHandle.IsTypeAndStorage<
                              vtkm::Float32,
                              vtkm::cont::internal::StorageTagError>();
```

Once the type of the DynamicArrayHandle is known, it can be cast to a concrete ArrayHandle, which has access to the data as described in Section 6.2. The easiest way to do this is to use the CopyTo method. This templated method takes a reference to an ArrayHandle as an argument and sets that array handle to point to the array in DynamicArrayHandle. If the given types are incorrect, then CastToArrayHandle throws a vtkm::cont::ErrorControlBadValue exception.

Example 6.41: Casting a DynamicArrayHandle to a concrete ArrayHandle.

```
dynamicHandle.CopyTo(concreteHandle);
```

6.3.2 Casting to Unknown Types

Using CastToArrayHandle is fine as long as the correct types are known, but often times they are not. For this use case DynamicArrayHandle has a method named CastAndCall that attempts to cast the array to some set of types.

The CastAndCall method accepts a functor to run on the appropriately cast array. The functor must have an overloaded const parentheses operator that accepts an ArrayHandle of the appropriate type.

Example 6.42: Operating on DynamicArrayHandle with CastAndCall.

```
struct PrintArrayContentsFunctor
{
  template<typename T, typename Storage>
  VTKM_CONT_EXPORT
  void operator()(const vtkm::cont::ArrayHandle<T,Storage> &array) const
  {
    this->PrintArrayPortal(array.GetPortalConstControl());
  }

private:
  template<typename PortalType>
  VTKM_CONT_EXPORT
  void PrintArrayPortal(const PortalType &portal) const
  {
    for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
    {
      // All ArrayPortal objects have ValueType for the type of each value.
      typedef typename PortalType::ValueType ValueType;

      ValueType value = portal.Get(index);

      vtkm::IdComponent numComponents =
          vtkm::VecTraits<ValueType>::GetNumberOfComponents(value);
      for (vtkm::IdComponent componentIndex = 0;
           componentIndex < numComponents;
           componentIndex++)
      {
        std::cout << " "
                  << vtkm::VecTraits<ValueType>::GetComponent(value,
                                                              componentIndex);
      }
      std::cout << std::endl;
    }
  }
};

template<typename DynamicArrayType>
void PrintArrayContents(const DynamicArrayType &array)
{
  array.CastAndCall(PrintArrayContentsFunctor());
}
```

6.3.3 Specifying Cast Lists

The CastAndCall method can only check a finite number of types. The default form of CastAndCall uses a default set of common types. These default lists can be overridden using the VTK-m list tags facility, which is discussed at length in Section 2.5. There are separate lists for component types and for storage types.

Common type lists for components are defined in vtkm/TypeListTag.h and are documented in Section 2.5.2. This header also defines VTKM_DEFAULT_TYPE_LIST_TAG, which defines the default list of component types tried in CastAndCall.

Common storage lists are defined in vtkm/cont/StorageListTag.h. There is only one common storage distributed with VTK-m: StorageBasic. A list tag containing this type is defined as vtkm::cont::StorageListTagBasic.

As with other lists, it is possible to create new storage type lists using the existing type lists and the list bases from Section 2.5.1.

The vtkm/cont/StorageListTag.h header also defines a macro named VTKM_DEFAULT_STORAGE_LIST_TAG that defines a default list of types to use in classes like DynamicArrayHandle.

This list can be overridden by defining the `VTKM_DEFAULT_STORAGE_LIST_TAG` macro *before* any VTK-m headers are included. If included after a VTK-m header, the list is not likely to take effect. Do not ignore compiler warnings about the macro being redefined, which you will not get if defined correctly.

There is a form of `CastAndCall` that accepts tags for the list of component types and storage types. This can be used when the specific lists are known at the time of the call. However, when creating generic operations like the `PrintArrayContents` function in Example 6.42, passing these tags is inconvenient at best.

To address this use case, `DynamicArrayHandle` has a pair of methods named `ResetTypeList` and `ResetStorageList`. These methods return a new object with that behaves just like a `DynamicArrayHandle` with identical state except that the cast and call functionality uses the specified component type or storage type instead of the default. (Note that `PrintArrayContents` in Example 6.42 is templated on the type of `DynamicArrayHandle`. This is to accommodate using the objects from the `Reset*List` methods, which have the same behavior but different type names.)

So the default component type list contains a subset of the basic VTK-m types. If you wanted to accommodate more types, you could use `ResetTypeList`.

Example 6.43: Trying all component types in a `DynamicArrayHandle`.

```
  PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagAll()));
```

Likewise, if you happen to know a particular type of the dynamic array, that can be specified to reduce the amount of object code created by templates in the compiler.

Example 6.44: Specifying a single component type in a `DynamicArrayHandle`.

```
  PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagId()));
```

Storage type lists can be changed similarly.

Example 6.45: Specifying different storage types in a `DynamicArrayHandle`.

```
struct MyIdStorageList :
    vtkm::ListTagBase<
        vtkm::cont::StorageTagBasic,
        vtkm::cont::ArrayHandleIndex::StorageTag>
{ };

void PrintIds(vtkm::cont::DynamicArrayHandle array)
{
  PrintArrayContents(array.ResetStorageList(MyIdStorageList()));
}
```

Both the component type list and the storage type list can be modified by chaining these reset calls.

Example 6.46: Specifying both component and storage types in a `DynamicArrayHandle`.

```
  PrintArrayContents(dynamicArray.
                     ResetTypeList(vtkm::TypeListTagId()).
                     ResetStorageList(MyIdStorageList()));
```

6.4 Data Sets

A *data set*, implemented with the `vtkm::cont::DataSet` class, contains and manages the geometric data structures that VTK-m operates on. A data set comprises the following 3 data structures.

Cell Set A cell set describes topological connections. A cell set defines some number of points in space and how they connect to form cells, filled regions of space. A data set must have at least one cell set, but can have more than one cell set defined. This makes it possible to define groups of cells with different properties. For example, a simulation might model some subset of elements as boundary that contain properties the other elements do not. Another example is the representation of a molecule that requires atoms and bonds, each having very different properties associated with them.

Field A field describes numerical data associated with the topological elements in a cell set. The field is represented as an array, and each entry in the field array corresponds to a topological element (point, edge, face, or cell). Together the cell set topology and discrete data values in the field provide an interpolated function throughout the volume of space covered by the data set. A cell set can have any number of fields.

Coordinate System A coordinate system is a special field that describes the physical location of the points in a data set. Although it is most common for a data set to contain a single coordinate system, VTK-m supports data sets with no coordinate system such as abstract data structures like graphs that might not have positions in a space. `DataSet` also supports multiple coordinate systems for data that have multiple representations for position. For example, geospatial data could simultaneously have coordinate systems defined by 3D position, latitude-longitude, and any number of 2D projections.

6.4.1 Building Data Sets

Before we go into detail on the cell sets, fields, and coordinate systems that make up a data set in VTK-m, let us first discuss how to build a data set. One simple way to build a data set is to load data from a file using the `vtkm::io` module. Reading files is discussed in detail in Chapter 3.

This section describes building data sets of different types using a set of classes named `DataSetBuilder*`, which provide a convenience layer on top of `vtkm::cont::DataSet` to make it easier to create data sets.

Creating Uniform Grids

Uniform grids are meshes that have a regular array structure with points uniformly spaced parallel to the axes. Uniform grids are also sometimes called regular grids or images.

The `vtkm::cont::DataSetBuilderUniform` class can be used to easily create 2- or 3-dimensional uniform grids. `DataSetBuilderUniform` has several versions of a method named `Create` that takes the number of points in each dimension, the origin, and the spacing. The origin is the location of the first point of the data (in the lower left corner), and the spacing is the distance between points in the x, y, and z directions. The `Create` methods also take an optional name for the coordinate system and an optional name for the cell set.

The following example creates a `vtkm::cont::DataSet` containing a uniform grid of $101 \times 101 \times 26$ points.

Example 6.47: Creating a uniform grid.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet = dataSetBuilder.Create(vtkm::Id3(101, 101, 26));
```

If not specified, the origin will be at the coordinates $(0, 0, 0)$ and the spacing will be 1 in each direction. Thus, in the previous example the width, height, and depth of the mesh in physical space will be 100, 100, and 25, respectively, and the mesh will be centered at $(50, 50, 12.5)$. Let us say we actually want a mesh of the same dimensions, but we want the z direction to be stretched out so that the mesh will be the same size in each direction, and we want the mesh centered at the origin.

Example 6.48: Creating a uniform grid with custom origin and spacing.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet =
    dataSetBuilder.Create(
      vtkm::Id3(101, 101, 26),
      vtkm::Vec<vtkm::FloatDefault,3>(-50.0, -50.0, -50.0),
      vtkm::Vec<vtkm::FloatDefault,3>(1.0, 1.0, 4.0));
```

Creating Rectilinear Grids

A rectilinear grid is similar to a uniform grid except that a rectilinear grid can adjust the spacing between adjacent grid points. This allows the rectilinear grid to have tighter sampling in some areas of space, but the points are still constrained to be aligned with the axes and each other. The irregular spacing of a rectilinear grid is specified by providing a separate array each for the x, y, and z coordinates.

The `vtkm::cont::DataSetBuilderRectilinear` class can be used to easily create 2- or 3-dimensional rectilinear grids. `DataSetBuilderRectilinear` has several versions of a method named `Create` that takes these coordinate arrays and builds a `vtkm::cont::-DataSet` out of them. The arrays can be supplied as either standard C arrays or as std::vector objects, in which case the data in the arrays are copied into the DataSet. [Support ArrayHandle as well?]

The following example creates a vtkm::cont::DataSet containing a rectilinear grid with 201 × 201 × 101 points with different irregular spacing along each axis.

Example 6.49: Creating a rectilinear grid.

```
// Make x coordinates range from -4 to 4 with tighter spacing near 0.
std::vector<vtkm::Float32> xCoordinates;
for (vtkm::Float32 x = -2.0f; x <= 2.0f; x += 0.02f)
{
  xCoordinates.push_back(vtkm::CopySign(x*x, x));
}

// Make y coordinates range from 0 to 2 with tighter spacing near 2.
std::vector<vtkm::Float32> yCoordinates;
for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
{
  yCoordinates.push_back(vtkm::Sqrt(y));
}

// Make z coordinates rangefrom -1 to 1 with even spacing.
std::vector<vtkm::Float32> zCoordinates;
for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
{
  zCoordinates.push_back(z);
}

vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;

vtkm::cont::DataSet dataSet = dataSetBuilder.Create(xCoordinates,
                                                     yCoordinates,
                                                     zCoordinates);
```

Creating Explicit Meshes

An explicit mesh is an arbitrary collection of cells with arbitrary connections. It can have multiple different types of cells. Explicit meshes are also known as unstructured grids.

The cells of an explicit mesh are defined by providing the shape, number of indices, and the points that comprise it for each cell. These three things are stored in separate arrays. Figure 6.1 shows an example of an explicit mesh and the arrays that can be used to define it.

The vtkm::cont::DataSetBuilderExplicit class can be used to create data sets with explicit meshes. DataSetBuilderExplicit has several versions of a method named Create. Generally, these methods take the shapes, number of indices, and connectivity arrays as well as an array of point coordinates. These arrays can be given in std::vector objects, and the data are copied into the DataSet created.

The following example creates a mesh like the one shown in Figure 6.1.

Example 6.50: Creating an explicit mesh with DataSetBuilderExplicit.

```
// Array of point coordinates.
```

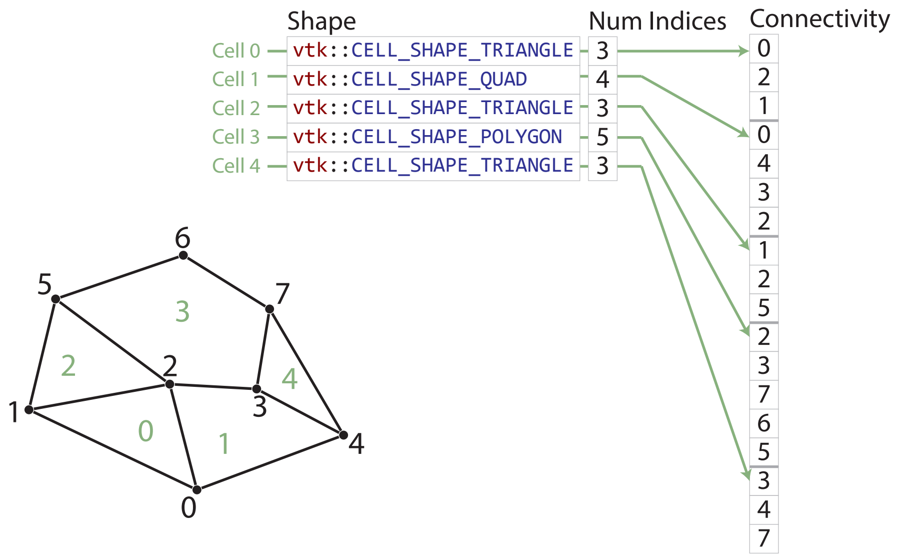


Figure 6.1: An example explicit mesh.

```
std::vector<vtkm::Vec<vtkm::Float32,3> > pointCoordinates;
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.1f, 0.0f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.2f, 0.4f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.9f, 0.6f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.4f, 0.5f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.8f, 0.3f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.4f, 1.0f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.0f, 1.2f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.5f, 0.9f, 0.0f));

// Array of shapes.
std::vector<vtkm::UInt8> shapes;
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
shapes.push_back(vtkm::CELL_SHAPE_QUAD);
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);

// Array of number of indices per cell.
std::vector<vtkm::IdComponent> numIndices;
numIndices.push_back(3);
numIndices.push_back(4);
numIndices.push_back(3);
numIndices.push_back(5);
numIndices.push_back(3);

// Connectivity array.
std::vector<vtkm::Id> connectivity;
connectivity.push_back(0); // Cell 0
connectivity.push_back(2);
connectivity.push_back(1);
connectivity.push_back(0); // Cell 1
connectivity.push_back(4);
connectivity.push_back(3);
connectivity.push_back(2);
connectivity.push_back(1); // Cell 2
connectivity.push_back(2);
connectivity.push_back(5);
connectivity.push_back(2); // Cell 3
connectivity.push_back(3);
connectivity.push_back(7);
connectivity.push_back(6);
connectivity.push_back(5);
connectivity.push_back(3); // Cell 4
connectivity.push_back(4);
connectivity.push_back(7);

// Copy these arrays into a DataSet.
vtkm::cont::DataSetBuilderExplicit dataSetBuilder;

vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointCoordinates,
                                                    shapes,
                                                    numIndices,
                                                    connectivity);
```

Often it is awkward to build your own arrays and then pass them to `DataSetBuilderExplicit`. There also exists an alternate builder class named `vtkm::cont::DataSetBuilderExplicitIterative` that allows you to specify each cell and point one at a time rather than all at once. This is done by calling one of the versions of `AddPoint` and one of the versions of `AddCell` for each point and cell, respectively. The next example also builds the mesh shown in Figure 6.1 except this time using `DataSetBuilderExplicitIterative`.

Example 6.51: Creating an explicit mesh with `DataSetBuilderExplicitIterative`.

```
vtkm::cont::DataSetBuilderExplicitIterative dataSetBuilder;

dataSetBuilder.AddPoint(1.1, 0.0, 0.0);
dataSetBuilder.AddPoint(0.2, 0.4, 0.0);
dataSetBuilder.AddPoint(0.9, 0.6, 0.0);
dataSetBuilder.AddPoint(1.4, 0.5, 0.0);
dataSetBuilder.AddPoint(1.8, 0.3, 0.0);
dataSetBuilder.AddPoint(0.4, 1.0, 0.0);
dataSetBuilder.AddPoint(1.0, 1.2, 0.0);
dataSetBuilder.AddPoint(1.5, 0.9, 0.0);

dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
dataSetBuilder.AddCellPoint(0);
dataSetBuilder.AddCellPoint(2);
dataSetBuilder.AddCellPoint(1);

dataSetBuilder.AddCell(vtkm::CELL_SHAPE_QUAD);
dataSetBuilder.AddCellPoint(0);
dataSetBuilder.AddCellPoint(4);
dataSetBuilder.AddCellPoint(3);
dataSetBuilder.AddCellPoint(2);

dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
dataSetBuilder.AddCellPoint(1);
dataSetBuilder.AddCellPoint(2);
dataSetBuilder.AddCellPoint(5);

dataSetBuilder.AddCell(vtkm::CELL_SHAPE_POLYGON);
dataSetBuilder.AddCellPoint(2);
dataSetBuilder.AddCellPoint(3);
dataSetBuilder.AddCellPoint(7);
dataSetBuilder.AddCellPoint(6);
dataSetBuilder.AddCellPoint(5);

dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
dataSetBuilder.AddCellPoint(3);
dataSetBuilder.AddCellPoint(4);
dataSetBuilder.AddCellPoint(7);

vtkm::cont::DataSet dataSet = dataSetBuilder.Create();
```

Add Fields

In addition to creating the geometric structure of a data set, it is usually important to add fields to the data. Fields describe numerical data associated with the topological elements in a cell. They often represent a physical quantity (such as temperature, mass, or volume fraction) but can also represent other information (such as indices or classifications).

The easiest way to define fields in a data set is to use the `vtkm::cont::DataSetFieldAdd` class. This class works on `DataSet`s of any type. It has methods named `AddPointField` and `AddCellField` that define a field for either points or cells. Every field must have an associated field name.

Both `AddPointField` and `AddCellField` are overloaded to accept arrays of data in different structures. Field arrays can be passed as standard C arrays or as `std::vector`s, in which case the data are copied. Field arrays can also be passed in a `ArrayHandle`, in which case the data are not copied.

The following (somewhat contrived) example defines fields for a uniform grid that identify which points and cells are on the boundary of the mesh.

Example 6.52: Adding fields to a `DataSet`.

```
// Make a simple structured data set.
const vtkm::Id3 pointDimensions(20, 20, 10);
const vtkm::Id3 cellDimensions = pointDimensions - vtkm::Id3(1, 1, 1);
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointDimensions);

// This is the helper object to add fields to a data set.
vtkm::cont::DataSetFieldAdd dataSetFieldAdd;

// Create a field that identifies points on the boundary.
std::vector<vtkm::UInt8> boundaryPoints;
for (vtkm::Id zIndex = 0; zIndex < pointDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < pointDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < pointDimensions[0]; xIndex++)
    {
      if ( (xIndex == 0) || (xIndex == pointDimensions[0]-1) ||
           (yIndex == 0) || (yIndex == pointDimensions[1]-1) ||
           (zIndex == 0) || (zIndex == pointDimensions[2]-1) )
      {
        boundaryPoints.push_back(1);
      }
      else
      {
        boundaryPoints.push_back(0);
      }
    }
  }
}

dataSetFieldAdd.AddPointField(dataSet, "boundary_points", boundaryPoints);

// Create a field that identifies cells on the boundary.
std::vector<vtkm::UInt8> boundaryCells;
for (vtkm::Id zIndex = 0; zIndex < cellDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < cellDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < cellDimensions[0]; xIndex++)
    {
      if ( (xIndex == 0) || (xIndex == cellDimensions[0]-1) ||
           (yIndex == 0) || (yIndex == cellDimensions[1]-1) ||
           (zIndex == 0) || (zIndex == cellDimensions[2]-1) )
      {
        boundaryCells.push_back(1);
      }
      else
      {
        boundaryCells.push_back(0);
      }
    }
  }
}

dataSetFieldAdd.AddCellField(dataSet, "boundary_cells", boundaryCells);
```

6.4.2 Cell Sets

A cell set determines the topological structure of the data in a data set. Fundamentally, any cell set is a collection of cells, which typically (but not always) represent some region in space. 3D cells are made up of points, edges, and faces. (2D cells have only points and edges, and 1D cells have only points.) The arrangement of these points, edges, and faces is defined by the *shape* of the cell, which prescribes a specific ordering of each. The basic cell shapes provided by VTK-m are discussed in detail in Section 8.3.1 starting on page 140.

There are multiple ways to express the connections of a cell set, each with different benefits and restrictions. These different cell set types are managed by different cell set classes in VTK-m. All VTK-m cell set classes inherit from `vtkm::cont::CellSet`. The two basic types of cell sets are structured and explicit, and there are several variations of these types.

Structured Cell Sets

A `vtkm::cont::CellSetStructured` defines a 1-, 2-, or 3-dimensional grid of points with lines, quadrilaterals, or hexahedra, respectively, connecting them. The topology of a `CellSetStructured` is specified by simply providing the dimensions, which is the number of points in the i, j, and k directions of the grid of points. The number of points is implicitly $i \times j \times k$ and the number of cells is implicitly $(i-1) \times (j-1) \times (k-1)$ (for 3D grids). Figure 6.2 demonstrates this arrangement.

The big advantage of using `vtkm::cont::CellSetStructured` to define a cell set is that it is very space efficient because the entire topology can be defined by the three integers specifying the dimensions. Also algorithms can be optimized for `CellSetStructured`'s regular nature. However, `CellSetStructured`'s strictly regular grid structure also limits its applicability. A structured cell set can only be a dense grid of lines, quadrilaterals, or hexahedra.

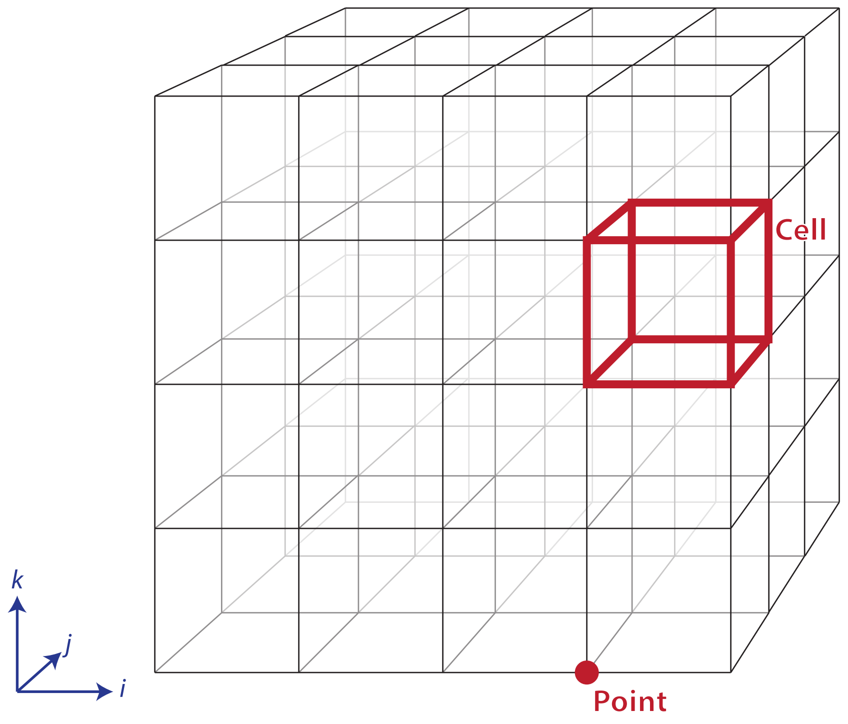


Figure 6.2: The arrangement of points and cells in a 3D structured grid.

It cannot represent irregular data well.

Many data models in other software packages, such as the one for VTK, make a distinction between uniform, rectilinear, and curvilinear grids. VTK-m's cell sets do not. All three of these grid types are represented by `CellSetStructured`. This is because in a VTK-m data set the cell set and the coordinate system are defined independently and used interchangeably. A structured cell set with uniform point coordinates makes a uniform grid. A structured cell set with point coordinates defined irregularly along coordinate axes makes a rectilinear grid. And a structured cell set with arbitrary point coordinates makes a curvilinear grid. The point coordinates are defined by the data set's coordinate system, which is discussed in Section 6.4.4 starting on page 86.

Explicit Cell Sets

A `vtkm::cont::CellSetExplicit` defines an irregular collection of cells. The cells can be of different types and connected in arbitrary ways. This is done by explicitly providing for each cell a sequence of points that defines the cell.

An explicit cell set is defined with a minimum of three arrays. The first array identifies the shape of each cell. (Cell shapes are discussed in detail in Section 8.3.1 starting on page 140.) The second array identifies how many points are in each cell. The third array has a sequence of point indices that make up each cell. Figure 6.3 shows a simple example

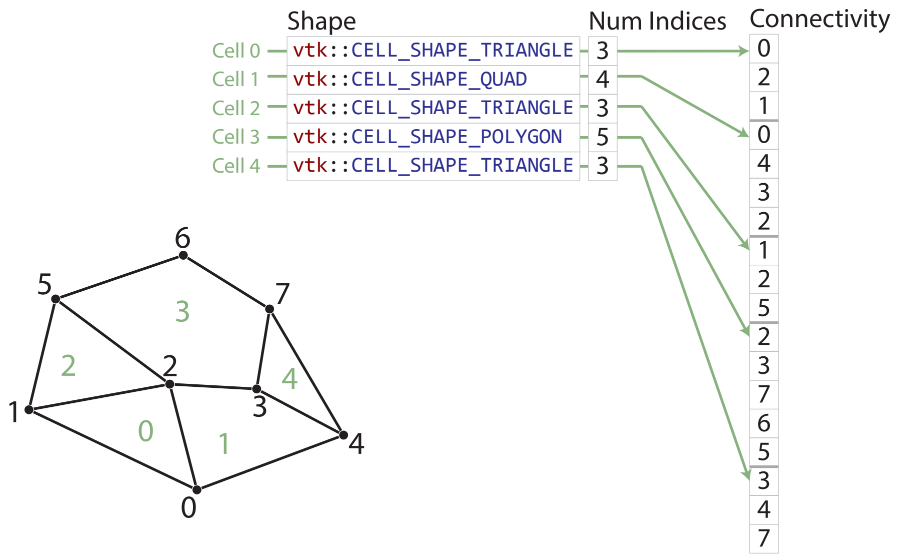


Figure 6.3: Example of cells in a CellSetExplict and the arrays that define them.

of an explicit cell set.

An explicit cell set may also have other topological arrays such as an array of offsets of each cell into the connectivity array or an array of cells incident on each point. Although these arrays can be provided, they are optional and can be internally derived from the shape, num indices, and connectivity arrays.

vtkm::cont::ExplicitCellSet is a powerful representation for a cell set because it can represent an arbitrary collection of cells. However, because all connections must be explicitly defined, ExplicitCellSet requires a significant amount of memory to represent the topology.

An important specialization of an explicit cell set is vtkm::cont::CellSetSingleType. CellSetSingleType is an explicit cell set constrained to contain cells that all have the same shape and all have the same number of points. So for example if you are creating a surface that you know will contain only triangles, CellSetSingleType is a good representation for these data.

Using CellSetSingleType saves memory because the array of cell shapes and the array of point counts no longer need to be stored. CellSetSingleType also allows VTK-m to skip some processing and other storage required for general explicit cell sets.

Cell Set Permutations

A vtkm::cont::CellSetPermutation rearranges the cells of one cell set to create another cell set. This restructuring of cells is not done by copying data to a new structure. Rather, CellSetPermutation establishes a look-up from one cell structure to another. Cells are permuted on the fly while algorithms are run.

A CellSetPermutation is established by providing a mapping array that for every cell index provides the equivalent cell index in the cell set being permuted. CellSetPermutation is most often used to mask out cells in a data set so that algorithms will skip over those cells when running. Note that although CellSetPermutation can mask cells, it cannot mask points. All points from the original cell set are available in the permuted cell set regardless of whether they are used.

The following example uses vtkm::cont::CellSetPermutation with a counting array to expose every tenth cell. This provides a simple way to subsample a data set.

Example 6.53: Subsampling a data set with CellSetPermutation.

```
// Create a simple data set.
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet originalDataSet =
    dataSetBuilder.Create(vtkm::Id3(33,33,26));
vtkm::cont::CellSetStructured<3> originalCellSet;
originalDataSet.GetCellSet().CopyTo(originalCellSet);

// Create a permutation array for the cells. Each value in the array refers
// to a cell in the original cell set. This particular array selects every
// 10th cell.
vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);

// Create a permutation of that cell set containing only every 10th cell.
vtkm::cont::CellSetPermutation<
      vtkm::cont::ArrayHandleCounting<vtkm::Id>,
      vtkm::cont::CellSetStructured<3> >
    permutedCellSet(permutationArray, originalCellSet);
```

Dynamic Cell Sets

vtkm::cont::DataSet must hold an arbitrary collection of vtkm::cont::CellSet objects, which it cannot do while knowing their types at compile time. To manage storing CellSets without knowing their types, DataSet actually holds references using vtkm::cont::DynamicCellSet.

DynamicCellSet is similar in nature to DynamicArrayHandle except that it, of course, holds CellSets instead of ArrayHandles. The interface for the two classes is similar, and you should review the documentation for DynamicArrayHandle (in Section 6.3 starting on page 70) to understand DynamicCellSet.

vtkm::cont::DynamicCellSet has a method named GetCellSet that returns a const reference to the held cell set as the abstract CellSet class. This can be used to easily access the virtual methods in the CellSet interface. You can also create a new instance of a cell set with the same type using the NewInstance method.

The DynamicCellSet::IsType() method can be used to determine whether the cell set held in the dynamic cell set is of a given type. If the cell set type is known, DynamicCellSet::CastTo() can be used to safely downcast the cell set object.

When a typed version of the cell set stored in the DynamicCellSet is needed but the type is not known, which happens regularly in the internal workings of VTK-m, the CastAndCall method can be used to make this transition. CastAndCall works by taking a functor and calls it with the appropriately cast cell set object.

The CastAndCall method works by attempting to cast to a known set of types. This set of types used is defined by the macro VTKM_DEFAULT_CELL_SET_LIST_TAG, which is declared in vtkm/cont/CellSetListTag.h. This list can be overridden globally by defining the VTKM_DEFAULT_CELL_SET_LIST_TAG macro *before* any VTK-m headers are included.

The set of types used in a CastAndCall can also be changed only for a particular instance of a dynamic cell set by calling its ResetCellSetList. This method takes a list of cell types and returns a new dynamic array handle of a slightly different type that will use this new list of cells for dynamic casting.

Blocks and Assemblies

Rather than just one cell set, a vtkm::cont::DataSet can hold multiple cell sets. This can be used to construct multiblock data structures or assemblies of parts. Multiple cell sets can also be used to represent subsets of the data with particular properties such as all cells filled with a material of a certain type. Or these multiple cells might represent particular features in the data, such as the set of faces representing a boundary in the simulation.

Zero Cell Sets

It is also possible to construct a vtkm::cont::DataSet that contains no cell set objects whatsoever. This can be used to manage data that does not contain any topological structure. For example, a collection of series that come from columns in a table could be stored as multiple fields in a data set with no cell set.

6.4.3 Fields

A field on a data set provides a value on every point in space on the mesh. Fields are often used to describe physical properties such as pressure, temperature, mass, velocity, and much more. Fields are represented in a VTK-m data set as an array where each value is associated with a particular element type of a mesh (such as points or cells). This association of field values to mesh elements and the structure of the cell set determines how the field is interpolated throughout the space of the mesh.

Fields are manged by the vtkm::cont::Field class. Field holds its data with a DynamicArrayHandle, which itself is a container for an ArrayHandle. Field also maintains the association and, optionally, the name of a cell set for which the field is valid.

The data array can be retrieved as a DynamicArrayHandle using the GetData method of Field. Field also has a convenience method named GetBounds that finds the range of values stored in the field array.

6.4.4 Coordinate Systems

A coordinate system determines the location of a mesh's elements in space. The spatial location is described by providing a 3D vector at each point that gives the coordinates there. The point coordinates can then be interpolated throughout the mesh.

Coordinate systems are managed by the vtkm::cont::CoordinateSystem class. In actuality, a coordinate system is just a field with a special meaning, and so the CoordinateSystem class inherits from the Field class. CoordinateSystem constrains the field to be associated with points and typically has 3D floating point vectors for values.

It is typical for a DataSet to have one coordinate system defined, but it is possible to define multiple coordinate systems. This is helpful when there are multiple ways to express coordinates. For example, positions in geographic may be expressed as Cartesian coordinates or as latitude-longitude coordinates. Both are valid and useful in different ways.

It is also valid to have a DataSet with no coordinate system. This is useful when the structure is not rooted in physical space. For example, if the cell set is representing a graph structure, there might not be any physical space that has meaning for the graph.

6.5 Timers

It is often the case that you need to measure the time it takes for an operation to happen. This could be for performing measurements for algorithm study or it could be to dynamically adjust scheduling.

Performing timing in a multi-threaded environment can be tricky because operations happen asynchronously. In the VTK-m control environment timing is simplified because the control environment operates on a single thread. However, operations invoked in the execution environment may run asynchronously to operations in the control environment.

To ensure that accurate timings can be made, VTK-m provides a vtkm::cont::Timer class that is templated on the device adapter to provide an accurate measurement of operations that happen on the device. If not template parameter is provided, the default device adapter is used.

The timing starts when the Timer is constructed. The time elapsed can be retrieved with a call to the GetElapsedTime method. This method will block until all operations in the execution environment complete so as to return an accurate time. The timer can be restarted with a call to the `Reset` method.

[THIS EXAMPLE NEEDS TO BE UPDATED WHEN SOMETHING INTERESTING CAN BE INVOKED.]

Example 6.54: Using `vtkm::cont::Timer`.

```
  vtkm::cont::ArrayHandle<vtkm::Float32> results;
//  vtkm::cont::DispatchMapField<vtkm::worklet::Elevation> dispatcher;

  vtkm::cont::Timer<> timer;

//  dispatcher.Invoke(grid.GetPointCoordinates(), results);

  // This call makes sure data is pulled back to the host in a host/device
  // architecture.
  results.GetPortalConstControl();

  vtkm::Float64 elapsedTime = timer.GetElapsedTime();

  std::cout << "Time to run: " << elapsedTime << std::endl;
```

6.6 Error Handling

VTK-m uses exceptions to report errors. All exceptions thrown by VTK-m will be a subclass of `vtkm::cont::Error`. For simple error reporting, it is possible to simply catch a `vtkm::cont::Error` and report the error message string reported by the `GetMessage` method.

Example 6.55: Simple error reporting.

```
int main(int argc, char **argv)
{
  try
  {
    // Do something cool with VTK-m
    // ...
  }
  catch (vtkm::cont::Error error)
  {
    std::cout << error.GetMessage() << std::endl;
    return 1;
  }
  return 0;
}
```

There are two subclasses to `vtkm::cont::Error`. These are `vtkm::cont::ErrorExecution` and `vtkm::cont::ErrorControl`, and they represent errors that happen in the respective execution and control environments.

Readers familiar with parallel programming will probably note the difficulty in raising errors in multi-threaded execution like what happens in the execution environment. In fact some devices, like CUDA devices, do not support exceptions at all. VTK-m handles the error reporting in the execution environment by flagging an error when it occurs and then throwing an error in the control environment after all threads have terminated. This means that the amount of execution that happens after an error is flagged is indeterminate and any output values should be considered incorrect.

The vtkm::cont::ErrorControl class is also broken down into several subclasses that can be independently caught to handle different types of errors. The following control errors exist and may be thrown.

vtkm::cont::ErrorControlAssert Thrown when an assertion fails, meaning a VTK-m operation reached an unexpected state. The header file vtkm/cont/Assert.h defines a macro named VTKM_ASSERT_CONT that behaves much like the POSIX C assert macro except that a ErrorControlAssert is thrown rather than killing the application outright.

vtkm::cont::ErrorControlBadAllocation Thrown when there is a problem accessing or manipulating memory. Often this is thrown when an allocation fails because there is insufficient memory, but other memory access errors can cause this to be thrown as well.

vtkm::cont::ErrorControlBadType Thrown when VTK-m attempts to perform an operation on an object that is of an incompatible type.

vtkm::cont::ErrorControlBadValue Thrown when a VTK-m function or method encounters an invalid value that inhibits progress.

vtkm::cont::ErrorControlInternal Thrown when VTK-m detects an internal state that should never be reached. This error usually indicates a bug in VTK-m or, at best, VTK-m failed to detect an invalid input it should have.

6.7 Device Adapter Algorithms

VTK-m comes with the templated class vtkm::cont::DeviceAdapterAlgorithm that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. The template has a single argument that specifies the device adapter tag.

Example 6.56: Prototype for vtkm::cont::DeviceAdapterAlgorithm.

```
namespace vtkm {
namespace cont {

template<typename DeviceAdapterTag>
struct DeviceAdapterAlgorithm;

}
} // namespace vtkm::cont
```

DeviceAdapterAlgorithm contains no state. It only has a set of static methods that implement its algorithms. The following methods are available.

Copy Copies data from an input array to an output array. The copy takes place in the execution environment.

LowerBounds The LowerBounds method takes three arguments. The first argument is an ArrayHandle of sorted values. The second argument is another ArrayHandle of items to find in the first array. LowerBounds find the index of the first item that is greater than or equal to the target value, much like the std::lower_bound STL algorithm. The results are returned in an ArrayHandle given in the third argument.

There are two specializations of LowerBounds. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an ArrayHandle of sorted vtkm::Ids and the second is an ArrayHandle of vtkm::Ids to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

Reduce The Reduce method takes an input array, initial value, and a binary function and computes a "total" of applying the binary function to all entries in the array. The provided binary function must be associative (but it need not be commutative). There is a specialization of Reduce that does not take a binary function and computes the sum.

ReduceByKey The ReduceByKey method works similarly to the Reduce method except that it takes an additional array of keys, which must be the same length as the values being reduced. The arrays are partitioned into segments that have identical adjacent keys, and a separate reduction is performed on each partition. The unique keys and reduced values are returned in separate arrays.

ScanInclusive The ScanInclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is the same as the first value in the input. The second value in the output is the sum of the first two values in the input. The third value in the output is the sum of the first three values of the input, and so on. ScanInclusive returns the sum of all values in the input. There are two forms of ScanInclusive: one performs the sum using addition whereas the other accepts a custom binary function to use as the sum operator.

ScanExclusive The ScanExclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is always 0. The second value in the output is the same as the first value in the input. The third value in the output is the sum of the first two values in the input. The fourth value in the output is the sum of the first three values of the input, and so on. ScanExclusive returns the sum of all values in the input. There are two forms of ScanExclusive: one performs the sum using addition whereas the other accepts a custom binary function to use as the sum operator and a custom initial value to use in the running sum.

Schedule The Schedule method takes a functor as its first argument and invokes it a number of times specified by the second argument. It should be assumed that each invocation of the functor occurs on a separate thread although in practice there could be some thread sharing.

There are two versions of the `Schedule` method. The first version takes a `vtkm::Id` and invokes the functor that number of times. The second version takes a `vtkm::Id3` and invokes the functor once for every entry in a 3D array of the given dimensions.

The functor is expected to be an object with a const overloaded parentheses operator. The operator takes as a parameter the index of the invocation, which is either a `vtkm::Id` or a `vtkm::Id3` depending on what version of `Schedule` is being used. The functor must also subclass `vtkm::exec::FunctorBase`, which provides the error handling facilities for the execution environment. `FunctorBase` contains a public method named `RaiseError` that takes a message and will cause a `vtkm::cont::ErrorExecution` exception to be thrown in the control environment.

`Sort` The `Sort` method provides an unstable sort of an array. There are two forms of the `Sort` method. The first takes an `ArrayHandle` and sorts the values in place. The second takes an additional argument that is a functor that provides the comparison operation for the sort.

`SortByKey` The `SortByKey` method works similarly to the `Sort` method except that it takes two `ArrayHandle`s: an array of keys and a corresponding array of values. The sort orders the array of keys in ascending values and also reorders the values so they remain paired with the same key. Like `Sort`, `SortByKey` has a version that sorts by the default less-than operator and a version that accepts a custom comparison functor.

`StreamCompact` The `StreamCompact` method selectively removes values from an array. The first argument is an `ArrayHandle` to be compacted and the second argument is an `ArrayHandle` of equal size with flags indicating whether the corresponding input value is to be copied to the output. The third argument is an output `ArrayHandle` whose length is set to the number of true flags in the stencil and the passed values are put in order to the output array.

There is also a second form of `StreamCompact` that only has the stencil and output as arguments. In this version, the output gets the corresponding index of where the input should be taken from.

`Synchronize` The `Synchronize` method waits for any asynchronous operations running on the device to complete and then returns.

`Unique` The `Unique` method removes all duplicate values in an `ArrayHandle`. The method will only find duplicates if they are adjacent to each other in the array. The easiest way to ensure that duplicate values are adjacent is to sort the array first.

There are two versions of `Unique`. The first uses the equals operator to compare entries. The second accepts a binary functor to perform the comparisons.

`UpperBounds` The `UpperBounds` method takes three arguments. The first argument is an `ArrayHandle` of sorted values. The second argument is another `ArrayHandle` of items to find in the first array. `UpperBounds` find the index of the first item that is greater than to the target value, much like the `std::upper_bound` STL algorithm. The results are returned in an `ArrayHandle` given in the third argument.

There are two specializations of UpperBounds. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an ArrayHandle of sorted vtkm::Ids and the second is an ArrayHandle of vtkm::Ids to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

6.8 Implementing Device Adapters

VTK-m comes with several implementations of device adapters so that it may be ported to a variety of platforms. It is also possible to provide new device adapters to support yet more devices, compilers, and libraries. A new device adapter provides a tag, a class to manage arrays in the execution environment, a collection of algorithms that run in the execution environment, and (optionally) a timer.

Most device adapters are associated with some type of device or library, and all source code related directly to that device is placed in a subdirectory of vtkm/cont. For example, files associated with CUDA are in vtkm/cont/cuda and files associated with the Intel Threading Building Blocks (TBB) are located in vtkm/cont/tbb. The documentation here assumes that you are adding a device adapter to the VTK-m source code and following these file conventions. However, it is also possible to define a device adapter outside of the core VTK-m, in which case the file paths might be different.

For the purposes of discussion in this section, we will give a simple example of implementing a device adapter using the std::thread class provided by C++11. We will call our device Cxx11Thread and place it in the directory vtkm/cont/cxx11.

By convention the implementation of device adapters within VTK-m are divided into 3 header files with the names DeviceAdapterTag∗.h, ArrayManagerExecution∗.h and DeviceAdapterAlgorithm∗.h, which are hidden in internal directories. The DeviceAdapter∗.h that most code includes is a trivial header that simply includes these other three files. For our example std::thread device, we will create the base header at vtkm/cont/cxx11/DeviceAdapterCxx11Thread.h. The contents are the following (with minutia like include guards removed).

Example 6.57: Contents of the base header for a device adapter.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h>
```

The reason VTK-m breaks up the code for its device adapters this way is that there is an interdependence between the implementation of each device adapter and the mechanism to pick a default device adapter. Breaking up the device adapter code in this way maintains an acyclic dependence among header files.

6.8.1 Tag

The device adapter tag, as described in Section 6.1 is a simple empty type that is used as a template parameter to identify the device adapter. Every device adapter implementation provides one. The device adapter tag is typically defined in an internal header file with a prefix of DeviceAdapterTag.

The device adapter tag should be created with the macro `VTKM_VALID_DEVICE_ADAPTER`. This adapter takes an abbreviated name that it will append to `DeviceAdapterTag` to make the tag structure. It will also create some support classes that allow VTK-m to introspect the device adapter. The macro also expects a unique integer identifier that is usually stored in a macro prefixed with `VTKM_DEVICE_ADAPTER_`. These identifiers for the device adapters provided by the core VTK-m are declared in vtkm/cont/internal/DeviceAdapterTag.h.

The following example gives the implementation of our custom device adapter, which by convention would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h header file.

Example 6.58: Implementation of a device adapter tag.

```
#include <vtkm/cont/internal/DeviceAdapterTag.h>

// If this device adapter were to be contributed to VTK-m, then this macro
// declaration should be moved to DeviceAdapterTag.h and given a unique
// number.
#define VTKM_DEVICE_ADAPTER_CXX11_THREAD 101

VTKM_VALID_DEVICE_ADAPTER(Cxx11Thread, VTKM_DEVICE_ADAPTER_CXX11_THREAD);
```

6.8.2 Array Manager Execution

VTK-m defines a template named `vtkm::cont::internal::ArrayManagerExecution` that is responsible for allocating memory in the execution environment and copying data between the control and execution environment. The execution array manager is typically defined in an internal header file with a prefix of ArrayManagerExecution.

Example 6.59: Prototype for `vtkm::cont::internal::ArrayManagerExecution`.

```
namespace vtkm {
namespace cont {
namespace internal {

template<typename T, typename StorageTag, typename DeviceAdapterTag>
class ArrayManagerExecution;

}
}
} // namespace vtkm::cont::internal
```

A device adapter must provide a partial specialization of `ArrayManagerExecution` for its device adapter tag. The implementation for `ArrayManagerExecution` is expected to manage the resources for a single array. All `ArrayManagerExecution` specializations must have a constructor that takes a pointer to a `vtkm::cont::internal::Storage` object. The `ArrayManagerExecution` should store a reference to this `Storage` object and use it to pass data between control and execution environments. Additionally, `ArrayManagerExecution` must provide the following elements.

`ValueType` A `typedef` of the type for each item in the array. This is the same type as the first template argument.

`PortalType` The type of an array portal that can be used in the execution environment to access the array.

`PortalConstType` A read-only (const) version of `PortalType`.

`GetNumberOfValues` A method that returns the number of values stored in the array. The results are undefined if the data has not been loaded or allocated.

`PrepareForInput` A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a `bool` flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a `PortalConstType` that points to the data.

`PrepareForInPlace` A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a `bool` flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a `PortalType` that points to the data.

`PrepareForOutput` A method that takes an array size and allocates an array in the execution environment of the specified size. The initial memory may be uninitialized. The method returns a `PortalType` to the data.

`RetrieveOutputData` This method takes a storage object, allocates memory in the control environment, and copies data from the execution environment into it. If the control and execution environments share arrays, then this can be a no-operation.

`CopyInto` This method takes an STL-compatible iterator and copies data from the execution environment into it.

`Shrink` A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked.

`ReleaseResources` A method that frees any resources (typically memory) in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying memory in methods such as PrepareForInput and RetrieveOutputData. This might require creating buffers in the control environment to efficiently move data from control array portals.

However, if the control and execution environments share the same memory space, the execution array manager can, and should, delegate all of its operations to the Storage it is constructed with. VTK-m comes with a class called vtkm::cont::internal::ArrayManagerExecutionShareWithControl that provides the implementation for an execution array manager that shares a memory space with the control environment. In this case, making the ArrayManagerExecution specialization be a trivial subclass is sufficient. Continuing our example of a device adapter based on C++11's std::thread class, here is the implementation of ArrayManagerExecution, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 6.60: Specialization of ArrayManagerExecution.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecution.h>
#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace vtkm {
namespace cont {
namespace internal {

template<typename T, typename StorageTag>
class ArrayManagerExecution<
      T, StorageTag, vtkm::cont::DeviceAdapterTagCxx11Thread>
    : public vtkm::cont::internal::ArrayManagerExecutionShareWithControl<
        T, StorageTag>
{
  typedef vtkm::cont::internal::ArrayManagerExecutionShareWithControl
      <T, StorageTag> Superclass;

public:
  VTKM_CONT_EXPORT
  ArrayManagerExecution(typename Superclass::StorageType *storage)
    : Superclass(storage) {  }
};

}
}
} // namespace vtkm::cont::internal
```

6.8.3 Algorithms

A device adapter implementation must also provide a specialization of vtkm::cont::DeviceAdapterAlgorithm, which is documented in Section 6.7. The implementation for the device adapter algorithms is typically placed in a header file with a prefix of DeviceAdapterAlgorithm.

Although there are many methods in DeviceAdapterAlgorithm, it is seldom necessary to implement them all. Instead, VTK-m comes with vtkm::cont::internal::DeviceAdapterAlgorithmGeneral that provides generic implementation for most of the required algorithms. By deriving the specialization of DeviceAdapterAlgorithm from DeviceAdapterAlgorithmGeneral, only the implementations for Schedule and Synchronize need to be implemented. All other algorithms can be derived from those.

That said, not all of the algorithms implemented in DeviceAdapterAlgorithmGeneral are optimized for all types of devices. Thus, it is worthwhile to provide algorithms optimized for the specific device when possible. In particular, it is best to provide specializations for the sort, scan, and reduce algorithms.

One point to note when implementing the Schedule methods is to make sure that errors handled in the execution environment are handled correctly. As described in Section 8.1, errors are signaled in the execution environment by calling RaiseError on a functor or worklet object. This is handled internally by the vtkm::exec::internal::ErrorMessageBuffer class. ErrorMessageBuffer really just holds a small string buffer, which must be provided by the device adapter's Schedule method.

So, before Schedule executes the functor it is given, it should allocate a small string array in the execution environment, initialize it to the empty string, encapsulate the array in an ErrorMessageBuffer object, and set this buffer object in the functor. When the execution completes, Schedule should check to see if an error exists in this buffer and throw a vtkm::cont::ErrorExecution if an error has been reported.

The following example is a minimal implementation of device adapter algorithms using C++11's std::thread class. Note that no attempt at providing optimizations has been attempted (and many are possible). By convention this code would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h header file.

Example 6.61: Minimal specialization of DeviceAdapterAlgorithm.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/DeviceAdapterAlgorithm.h>
#include <vtkm/cont/internal/DeviceAdapterAlgorithmGeneral.h>

#include <thread>

namespace vtkm {
namespace cont {

template<>
struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
    : vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
          DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>,
          vtkm::cont::DeviceAdapterTagCxx11Thread>
{
private:
  template<typename FunctorType>
  struct ScheduleKernel1D
  {
    VTKM_CONT_EXPORT
    ScheduleKernel1D(const FunctorType &functor)
      : Functor(functor)
    {  }
    VTKM_EXEC_EXPORT
    void operator()() const
    {
      for (vtkm::Id threadId = this->BeginId;
           threadId < this->EndId;
           threadId++)
      {
        this->Functor(threadId);
        // If an error is raised, abort execution.
        if (this->ErrorMessage.IsErrorRaised()) { return; }
      }
    }

    FunctorType Functor;
    vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
    vtkm::Id BeginId;
    vtkm::Id EndId;
  };

  template<typename FunctorType>
  struct ScheduleKernel3D
  {
    VTKM_CONT_EXPORT
    ScheduleKernel3D(const FunctorType &functor, vtkm::Id3 maxRange)
      : Functor(functor), MaxRange(maxRange)
    {  }

    VTKM_EXEC_EXPORT
    void operator()() const
    {
      vtkm::Id3 threadId3D(this->BeginId%this->MaxRange[0],
                           (this->BeginId/this->MaxRange[0])%this->MaxRange[1],
                           this->BeginId/(this->MaxRange[0]*this->MaxRange[1]));

      for (vtkm::Id threadId = this->BeginId;
           threadId < this->EndId;
           threadId++)
      {
        this->Functor(threadId3D);
        // If an error is raised, abort execution.
        if (this->ErrorMessage.IsErrorRaised()) { return; }

        threadId3D[0]++;
        if (threadId3D[0] >= MaxRange[0])
        {
          threadId3D[0] = 0;
          threadId3D[1]++;
          if (threadId3D[1] >= MaxRange[1])
          {
            threadId3D[1] = 0;
            threadId3D[2]++;
          }
        }
      }
    }

    FunctorType Functor;
    vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
    vtkm::Id BeginId;
    vtkm::Id EndId;
    vtkm::Id3 MaxRange;
  };

  template<typename KernelType>
  VTKM_CONT_EXPORT
  static void DoSchedule(KernelType kernel,
                         vtkm::Id numInstances)
  {
    if (numInstances < 1) { return; }

    const vtkm::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    vtkm::exec::internal::ErrorMessageBuffer errorMessage(errorString,
                                                          MESSAGE_SIZE);
    kernel.Functor.SetErrorMessageBuffer(errorMessage);
    kernel.ErrorMessage = errorMessage;

    vtkm::Id numThreads =
        static_cast<vtkm::Id>(std::thread::hardware_concurrency());
    if (numThreads > numInstances)
    {
      numThreads = numInstances;
    }
    vtkm::Id numInstancesPerThread = (numInstances+numThreads-1)/numThreads;

    std::thread *threadPool = new std::thread[numThreads];
    vtkm::Id beginId = 0;
    for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
    {
      vtkm::Id endId = std::min(beginId+numInstancesPerThread, numInstances);
      KernelType threadKernel = kernel;
      threadKernel.BeginId = beginId;
      threadKernel.EndId = endId;
      std::thread newThread(threadKernel);
      threadPool[threadIndex].swap(newThread);
      beginId = endId;
    }

    for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
    {
      threadPool[threadIndex].join();
    }

    delete[] threadPool;

    if (errorMessage.IsErrorRaised())
    {
      throw vtkm::cont::ErrorExecution(errorString);
    }
  }

public:
  template<typename FunctorType>
  VTKM_CONT_EXPORT
  static void Schedule(FunctorType functor, vtkm::Id numInstances)
  {
    DoSchedule(ScheduleKernel1D<FunctorType>(functor), numInstances);
  }

  template<typename FunctorType>
  VTKM_CONT_EXPORT
  static void Schedule(FunctorType functor, vtkm::Id3 maxRange)
  {
    vtkm::Id numInstances = maxRange[0]*maxRange[1]*maxRange[2];
    DoSchedule(ScheduleKernel3D<FunctorType>(functor, maxRange), numInstances);
  }

  VTKM_CONT_EXPORT
  static void Synchronize()
  {
    // Nothing to do. This device schedules all of its operations using a
    // split/join paradigm. This means that the if the control threaad is
    // calling this method, then nothing should be running in the execution
    // environment.
  }
```

```
};

}
} // namespace vtkm::cont
```

6.8.4 Timer Implementation

The VTK-m timer, described in Section 6.5, delegates to an internal class named vtkm::cont::DeviceAdapterTimerImplementation. The interface for this class is the same as that for vtkm::cont::Timer. A default implementation of this templated class uses the system timer and the Synchronize method in the device adapter algorithms.

However, some devices might provide alternate or better methods for implementing timers. For example, the TBB and CUDA libraries come with high resolution timers that have better accuracy than the standard system timers. Thus, the device adapter can optionally provide a specialization of DeviceAdapterTimerImplementation, which is typically placed in the same header file as the device adapter algorithms.

Continuing our example of a custom device adapter using C++11's std::thread class, we could use the default timer and it would work fine. But C++11 also comes with a std::chrono package that contains some portable time functions. The following code demonstrates creating a custom timer for our device adapter using this package. By convention, DeviceAdapterTimerImplementation is placed in the same header file as DeviceAdapterAlgorithm.

Example 6.62: Specialization of DeviceAdapterTimerImplementation.

```
#include <chrono>

namespace vtkm {
namespace cont {

template<>
class DeviceAdapterTimerImplementation<vtkm::cont::DeviceAdapterTagCxx11Thread>
{
public:
  VTKM_CONT_EXPORT
  DeviceAdapterTimerImplementation()
  {
    this->Reset();
  }

  VTKM_CONT_EXPORT
  void Reset()
  {
    vtkm::cont::DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
        ::Synchronize();
    this->StartTime = std::chrono::high_resolution_clock::now();
  }

  VTKM_CONT_EXPORT
  vtkm::Float64 GetElapsedTime()
  {
    vtkm::cont::DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
        ::Synchronize();
    std::chrono::high_resolution_clock::time_point endTime =
        std::chrono::high_resolution_clock::now();

    std::chrono::high_resolution_clock::duration elapsedTicks =
        endTime - this->StartTime;

    std::chrono::duration<vtkm::Float64> elapsedSeconds(elapsedTicks);

    return elapsedSeconds.count();
  }

private:
  std::chrono::high_resolution_clock::time_point StartTime;
};

}
} // namespace vtkm::cont
```

Chapter 7

Worklets

The simplest way to implement an algorithm in VTK-m is to create a *worklet*. A worklet is fundamentally a functor that operates on an element of data. Thus, it is a `class` or `struct` that has an overloaded parenthesis operator (which must be declared `const` for thread safety). However, worklets are also embedded with a significant amount of metadata on how the data should be managed and how the execution should be structured. This chapter explains the basic mechanics of defining and using worklets.

7.1 Worklet Types

Different operations in visualization can have different data access patterns, perform different execution flow, and require different provisions. VTK-m manages these different accesses, execution, and provisions by grouping visualization algorithms into common classes of operation and supporting each class with its own worklet type.

Each worklet type has a generic superclass that worklets of that particular type must inherit. This makes the type of the worklet easy to identify. The following list describes each worklet type provided by VTK-m and the superclass that supports it. Details on how to create worklets of each type are given in Section 7.5. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 9.

Field Map A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over a mesh, a `WorkletMapField` may actually be applied to any array. Thus, a field map can be used as a basic map operation.

Topology Map A worklet deriving `vtkm::worklet::WorkletMapTopology` or one of its sibling classes performs a mapping operation that applies a function (the operator in the worklet) on all elements of a particular type (such as points or cells) and creates a new field for those elements. The basic operation is similar to a field map except that in addition to access fields being mapped on, the worklet operation also has access to incident fields.

There are multiple convenience classes available for the most common types of topology mapping. `vtkm::worklet::WorkletMapPointToCell` calls the worklet operation for each cell and makes every incident point available. This type of map also has access to cell structures and can interpolate point fields.

7.2 Dispatchers

Worklets, both those provided by VTK-m as listed in Section 7.3 and ones created by a user as described in Section 7.4, are instantiated in the control environment and run in the execution environment. This means that the control environment must have a means to *invoke* worklets that start running in the execution environment.

This invocation is done through a set of *dispatcher* objects. A dispatcher object is an object in the control environment that has an instance of a worklet and can invoke that worklet with a set of arguments. There are multiple types of dispatcher objects, each corresponding to a type of worklet object. All dispatcher objects have at least two template parameters: the worklet class being invoked, which is always the first argument, and the device adapter tag, which is always the last argument and will be set to the default device adapter if not specified.

All dispatcher classes have a method named `Invoke` that launches the worklet in the execution environment. The arguments to `Invoke` must match those expected by the worklet, which is specified by something called a *control signature*. The expected arguments for worklets provided by VTK-m are documented in Section 7.3. Also, for any worklet, the `Invoke` arguments can be gleaned from the control signature, which is described in Section 7.4.1.

The following is a list of the dispatchers defined in VTK-m. The dispatcher classes correspond to the list of worklet types specified in Section 7.1. Many examples of using these dispatchers are provided in Section 7.3.

`vtkm::worklet::DispatcherMapField` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapField`. The dispatcher class has two template arguments: the worklet type and the device adapter (optional).

`vtkm::worklet::DispatcherMapTopology` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapTopology` or one of its sibling classes (such as `vtkm::worklet::WorkletMapPointToCell`). The dispatcher class has two template arguments: the worklet type and the device adapter (optional).

7.3 Provided Worklets

[Write this once some worklets are provided.]

7.4 Creating Worklets

A worklet is created by implementing a `class` or `struct` with the following features.

1. The class must contain a `ControlSignature` `typedef`, which specifies what arguments are expected when invoking the class with a dispatcher in the control environment.
2. The class must contain an `ExecutionSignature` `typedef`, which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.
3. The class must contain an `InputDomain` `typedef`, which identifies which input parameter defines the input domain of the data.
4. The class may define a scatter operation to override a 1:1 mapping from input to output.
5. The class must contain an overload of the parenthesis operator, which is the method that is executed in the execution environment.
6. The class must publicly inherit from a base worklet class that specifies the type of operation being performed.

Figure 7.1 demonstrates all of the required components of a worklet.

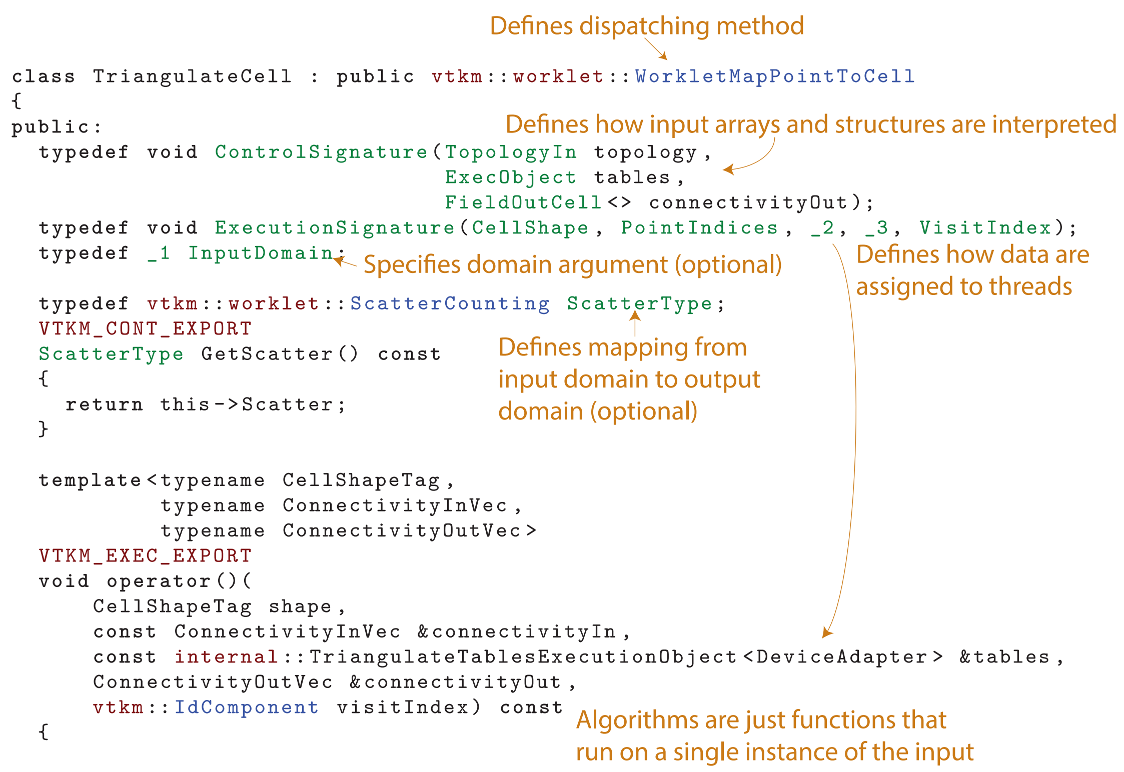
7.4.1 Control Signature

The control signature of a worklet is the `typedef` of a function prototype named `ControlSignature`. The function prototype matches the calling specification used with the dispatcher `Invoke` function.

Example 7.1: A `ControlSignature`.

```
typedef void ControlSignature(FieldIn<VecAll> inputVectors,
                              FieldOut<Scalar> outputMagnitudes);
```

The return type of the function prototype is always `void` because the dispatcher `Invoke` functions do not return values. The parameters of the function prototype are *tags* that


Defines dispatching method
Defines how input arrays and structures are interpreted
Specifies domain argument (optional)
Defines how data are assigned to threads
Defines mapping from input domain to output domain (optional)
Algorithms are just functions that run on a single instance of the input

```
class TriangulateCell : public vtkm::worklet::WorkletMapPointToCell
{
public:
  typedef void ControlSignature(TopologyIn topology,
                                ExecObject tables,
                                FieldOutCell<> connectivityOut);
  typedef void ExecutionSignature(CellShape, PointIndices, _2, _3, VisitIndex);
  typedef _1 InputDomain;

  typedef vtkm::worklet::ScatterCounting ScatterType;
  VTKM_CONT_EXPORT
  ScatterType GetScatter() const
  {
    return this->Scatter;
  }

  template<typename CellShapeTag,
           typename ConnectivityInVec,
           typename ConnectivityOutVec>
  VTKM_EXEC_EXPORT
  void operator()(
      CellShapeTag shape,
      const ConnectivityInVec &connectivityIn,
      const internal::TriangulateTablesExecutionObject<DeviceAdapter> &tables,
      ConnectivityOutVec &connectivityOut,
      vtkm::IdComponent visitIndex) const
  {
```

Figure 7.1: Annotated example of a worklet declaration.

identify the type of data that is expected to be passed to invoke. `ControlSignature` tags are defined by the worklet type and the various tags are documented more fully in Section 7.5.

By convention, `ControlSignature` tag names start with the base concept (e.g. `Field` or `Topology`) followed by the domain (e.g. `Point` or `Cell`) followed by `In` or `Out`. For example, `FieldPointIn` would specify values for a field on the points of a mesh that are used as input (read only). Although they should be there in most cases, some tag names might leave out the domain or in/out parts if they are obvious or ambiguous.

Type List Tags

Some tags are templated to have modifiers. For example, `Field` tags have a template argument that is set to a type list tag defining what types of field data are supported. (See Section 2.5.2 for a description of type lists.) In fact, this type list modifier is so common that the following convenience subtags used with `Field` tags are defined for all worklet types.

`AllTypes` All possible types.

`CommonTypes` The most used types in visualization. This includes signed integers and floats that are 32 or 64 bit. It also includes 3 dimensional vectors of floats. The same as `vtkm::TypeListTagCommon`.

IdType Contains the single item vtkm::Id. The same as vtkm::TypeListTagId.

Id2Type Contains the single item vtkm::Id2. The same as vtkm::TypeListTagId2.

Id3Type Contains the single item vtkm::Id3. The same as vtkm::TypeListTagId3.

Index All types used to index arrays. Contains vtkm::Id, vtkm::Id2, and vtkm::Id3. The same as vtkm::TypeListTagIndex.

FieldCommon A list containing all the types generally used for fields. It is the combination of Scalar, Vec2, Vec3, and Vec4. The same as vtkm::TypeListTagField.

Scalar Types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. vtkm::Float32 and vtkm::Float64). The same as vtkm::TypeListTagFieldScalar.

ScalarAll All scalar types. It contains signed and unsigned integers of widths from 8 to 64 bits. It also contains floats of 32 and 64 bit widths. The same as vtkm::TypeListTagScalarAll.

Vec2 Types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers. The same as vtkm::TypeListTagFieldVec2.

Vec3 Types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers. The same as vtkm::TypeListTagFieldVec3.

Vec4 Types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers. The same as vtkm::TypeListTagFieldVec4.

VecAll All vtkm::Vec classes with standard integers or floating points as components and lengths between 2 and 4. The same as vtkm::TypeListTagVecAll.

VecCommon The most common vector types. It contains all vtkm::Vec class of size 2 through 4 containing components of unsigned bytes, signed 32-bit integers, signed 64-bit integers, 32-bit floats, or 64-bit floats. The same as vtkm::TypeListTagVecCommon.

7.4.2 Execution Signature

Like the control signature, the execution signature of a worklet is the `typedef` of a function prototype named ExecutionSignature. The function prototype must match the parenthesis operator (described in Section 7.4.4) in terms of arity and argument semantics.

Example 7.2: An ExecutionSignature.

```
typedef _2 ExecutionSignature(_1);
```

The arguments of the ExecutionSignature's function prototype are tags that define where the data come from. The most common tags are an underscore followed by a number, such as _1, _2, etc. These numbers refer back to the corresponding argument in the ControlSignature. For example, _1 means data from the first control signature argument, _2 means data from the second control signature argument, etc.

Unlike the control signature, the execution signature optionally can declare a return type if the parenthesis operator returns a value. If this is the case, the return value should be one of the numeric tags (i.e. _1, _2, etc.) to refer to one of the data structures of the control signature. If the parenthesis operator does not return a value, then ExecutionSignature should declare the return type as void.

In addition to the numeric tags, there are other execution signature tags to represent other types of data. For example, the WorkIndex tag identifies the instance of the worklet invocation. Each call to the worklet function will have a unique WorkIndex. Other such tags exist and are described in the following section on worklet types where appropriate.

7.4.3 Input Domain

All worklets represent data parallel operations that are executed over independent elements in some domain. The type of domain is inherent from the worklet type, but the size of the domain is dependent on the data being operated on. One of the arguments given to the dispatcher's Invoke in the control environment must specify the domain.

A worklet identifies the argument specifying the domain with a typedef named InputDomain. The InputDomain must be typedefed to one of the execution signature numeric tags (i.e. _1, _2, etc.). By default, the InputDomain points to the first argument, but a worklet can override that to point to any argument.

Example 7.3: An InputDomain declaration.

```
typedef _1 InputDomain;
```

Different types of worklets can have different types of domain. For example a simple field map worklet has a FieldIn argument as its input domain, and the size of the input domain is taken from the size of the associated field array. Likewise, a worklet that maps topology has a TopologyIn argument as its input domain, and the size of the input domain is taken from the cell set.

Specifying the InputDomain is optional. If it is not specified, the first argument is assumed to be the input domain.

7.4.4 Worklet Operator

A worklet is fundamentally a functor that operates on an element of data. Thus, the algorithm that the worklet represents is contained in or called from the parenthesis operator method.

Example 7.4: An overloaded parenthesis operator of a worklet.

```
template<typename T, vtkm::IdComponent Size>
VTKM_EXEC_EXPORT
T operator()(const vtkm::Vec<T,Size> &inVector) const
{
```

There are some constraints on the parenthesis operator. First, it must have the same arity as the `ExecutionSignature`, and the types of the parameters and return must be compatible. Second, because it runs in the execution environment, it must be declared with the `VTKM_EXEC_EXPORT` (or `VTKM_EXEC_CONT_EXPORT`) modifier. Third, the method must be declared `const` to help preserve thread safety.

7.5 Worklet Type Reference

There are multiple worklet types provided by VTK-m, each designed to support a particular type of operation. Section 7.1 gave a brief overview of each type of worklet. This section gives a much more detailed reference for each of the worklet types including identifying the generic superclass that a worklet instance should derive, listing the signature tags and their meanings, and giving an example of the worklet in use.

7.5.1 Field Map

A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over the mesh, a `WorkletMapField` can actually be applied to any array.

A `WorkletMapField` subclass is invoked with a `vtkm::worklet::DispatcherMapField`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A field map worklet supports the following tags in the parameters of its `ControlSignature`.

`FieldIn` This tag represents an input field. A `FieldIn` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`.

Each invocation of the worklet gets a single value out of this array.

`FieldIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

The worklet's `InputDomain` can be set to a `FieldIn` argument. In this case, the input domain will be the size of the array.

`FieldOut` This tag represents an output field. A `FieldOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldInOut` This tag represents field that is both an input and an output. A `FieldInOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`FieldInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

The worklet's `InputDomain` can be set to a `FieldInOut` argument. In this case, the input domain will be the size of the array.

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the dispatcher's `Invoke`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the dispatcher's `Invoke`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

`WholeArrayOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the dispatcher's `Invoke`. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

`WholeArrayInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`, and this same object is given to the worklet. Execution objects are discussed in detail in Section 7.7 starting on page 125.

A field map worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 7.8).

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 7.5: Implementation and use of a field map worklet.

```
#include <vtkm/worklet/DispatcherMapField.h>
#include <vtkm/worklet/WorkletMapField.h>

#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/DynamicArrayHandle.h>

#include <vtkm/VectorAnalysis.h>


class Magnitude : public vtkm::worklet::WorkletMapField
{
public:
  typedef void ControlSignature(FieldIn<VecAll> inputVectors,
                                FieldOut<Scalar> outputMagnitudes);
  typedef _2 ExecutionSignature(_1);

  typedef _1 InputDomain;

  template<typename T, vtkm::IdComponent Size>
  VTKM_EXEC_EXPORT
  T operator()(const vtkm::Vec<T,Size> &inVector) const
  {
    return vtkm::Magnitude(inVector);
  }
};

VTKM_CONT_EXPORT
vtkm::cont::DynamicArrayHandle
InvokeMagnitude(vtkm::cont::DynamicArrayHandle input)
{
  vtkm::cont::ArrayHandle<vtkm::Float64> output;

  vtkm::worklet::DispatcherMapField<Magnitude> dispatcher;
  dispatcher.Invoke(input, output);

  return vtkm::cont::DynamicArrayHandle(output);
}
```

Although simple, the `WorkletMapField` worklet type can be used (and abused) as a general parallel-for/scheduling mechanism. In particular, the `WorkIndex` execution signature tag can be used to get a unique index, the `WholeArray`* tags can be used to get random access to arrays, and the `ExecObject` control signature tag can be used to pass execution objects directly to the worklet. Whole arrays and execution objects are talked about in more detail in Sections 7.6 and 7.7, respectively, in more detail, but here is a simple example that uses the random access of `WholeArrayOut` to make a worklet that copies an array in reverse order.

Example 7.6: Leveraging field maps and field maps for general processing.

```
struct ReverseArrayCopy : vtkm::worklet::WorkletMapField
{
  typedef void ControlSignature(FieldIn<> inputArray,
                                WholeArrayOut<> outputArray);
  typedef void ExecutionSignature(_1, _2, WorkIndex);
  typedef _1 InputDomain;

  template<typename InputType, typename OutputArrayPortalType>
  VTKM_EXEC_EXPORT
  void operator()(const InputType &inputValue,
                  const OutputArrayPortalType &outputArrayPortal,
                  vtkm::Id workIndex) const
  {
    vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
    if (outIndex >= 0)
    {
      outputArrayPortal.Set(outIndex, inputValue);
    }
    else
    {
      this->RaiseError("Output array not sized correctly.");
    }
  }
};

template<typename T, typename Storage>
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<T>
InvokeReverseArrayCopy(const vtkm::cont::ArrayHandle<T,Storage> &inArray)
{
  vtkm::cont::ArrayHandle<T> outArray;
  outArray.Allocate(inArray.GetNumberOfValues());

  vtkm::worklet::DispatcherMapField<ReverseArrayCopy> dispatcher;
  dispatcher.Invoke(inArray, outArray);

  return outArray;
}
```

7.5.2 Topology Map

A topology map performs a mapping that it applies a function (the operator in the worklet) on all the elements of a `DataSet` of a particular type (i.e. point, edge, face, or cell). While operating on the element, the worklet has access to data from all incident elements of another type.

There are several versions of topology maps that differ in what type of element being mapped from and what type of element being mapped to. The subsequent sections describe these different variations of the topology maps. Regardless of their names, they are all defined in vtkm/worklet/WorkletMapTopology.h and are all invoked with `vtkm::worklet::DispatcherMapTopology`.

Point to Cell Map

A worklet deriving `vtkm::worklet::WorkletMapPointToCell` performs a mapping operation that applies a function (the operator in the worklet) on all the cells of a `DataSet`. While operating on the cell, the worklet has access to fields associated both with the cell and with all incident points. Additionally, the worklet can get information about the structure of the cell and can perform operations like interpolation on it.

A `WorkletMapPointToCell` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A point to cell map worklet supports the following tags in the parameters of its `ControlSignature`.

`TopologyIn` This tag represents the cell set that defines the collection of cells the map will operate on. A `TopologyIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Section 8.3.)

There must be exactly one `TopologyIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of points.

Each invocation of the worklet gets a Vec-like object containing the field values for all the points incident with the cell being visited. The order of the entries is consistent with the defined order of the vertices for the visited cell's shape. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

`FieldInCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldOutCell` This tag represents an output field, which is necessarily associated with cells. A `FieldOutCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOutCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldOut` is an alias for `FieldOutCell` (since output arrays can only be defined on cells).

`FieldInOutCell` This tag represents field that is both an input and an output, which is necessarily associated with cells. A `FieldInOutCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`FieldInOutCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldInOut` is an alias for `FieldInOutCell` (since output arrays can only be defined on cells).

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the dispatcher's `Invoke`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase, and this same object is given to the worklet. Execution objects are discussed in detail in Section 7.7 starting on page 125.

A field map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

CellShape This tag produces a shape tag corresponding to the shape of the visited cell. (Cell shapes and the operations you can do with cells are discussed in Section 8.3.) This is the same value that gets provided if you reference the TopologyIn parameter.

PointCount This tag produces a vtkm::IdComponent equal to the number of points incident on the cell being visited. The Vecs provided from a FieldInPoint parameter will be the same size as PointCount.

PointIndices This tag produces a Vec-like object of vtkm::Ids giving the indices for all incident points. Like values from a FieldInPoint parameter, the order of the entries is consistent with the defined order of the vertices for the visited cell's shape.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 7.8).

Point to cell field maps are a powerful construct that allow you to interpolate point fields throughout the space of the data set. The following example provides a simple demonstration that finds the geometric center of each cell by interpolating the point coordinates to the cell centers.

Example 7.7: Implementation and use of a map point to cell worklet.

```
#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/cont/DataSet.h>
#include <vtkm/cont/DataSetFieldAdd.h>

#include <vtkm/exec/CellInterpolate.h>
#include <vtkm/exec/ParametricCoordinates.h>


class CellCenter : public vtkm::worklet::WorkletMapPointToCell
{
public:
  typedef void ControlSignature(TopologyIn cellSet,
                                FieldInPoint<> inputPointField,
                                FieldOut<> outputCellField);
  typedef _3 ExecutionSignature(_1, PointCount, _2);

  typedef _1 InputDomain;

  template<typename CellShape,
           typename InputPointFieldType>
  VTKM_EXEC_EXPORT
  typename InputPointFieldType::ComponentType
  operator()(CellShape shape,
             vtkm::IdComponent numPoints,
             const InputPointFieldType &inputPointField) const
  {
    vtkm::Vec<vtkm::FloatDefault,3> parametricCenter =
        vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, *this);
    return vtkm::exec::CellInterpolate(inputPointField,
                                       parametricCenter,
                                       shape,
                                       *this);
  }
};

VTKM_CONT_EXPORT
void FindCellCenters(vtkm::cont::DataSet &dataSet)
{
  vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault,3> > cellCentersArray;

  vtkm::worklet::DispatcherMapTopology<CellCenter> dispatcher;
  dispatcher.Invoke(dataSet.GetCellSet(),
                    dataSet.GetCoordinateSystem().GetData(),
                    cellCentersArray);

  vtkm::cont::DataSetFieldAdd dataSetFieldAdd;
  dataSetFieldAdd.AddCellField(dataSet, "cell_center", cellCentersArray);
}
```

Cell To Point Map

A worklet deriving `vtkm::worklet::WorkletMapCellToPoint` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `DataSet`. While operating on the point, the worklet has access to fields associated both with the point and with all incident cells.

A `WorkletMapCellToPoint` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A cell to point map worklet supports the following tags in the parameters of its `ControlSignature`.

`TopologyIn` This tag represents the cell set that defines the collection of points the map will operate on. A `TopologyIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`.

There must be exactly one `TopologyIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of cells.

Each invocation of the worklet gets a Vec-like object containing the field values for all the cells incident with the point being visited. The order of the entries is arbitrary but will be consistent with the values of all other `FieldInCell` arguments for the same worklet invocation. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of points. Each invocation of the worklet gets a single value out of this array.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldOutPoint` This tag represents an output field, which is necessarily associated with points. A `FieldOutPoint` argument expects an `ArrayHandle` or a `DynamicArrayHan-`

dle in the associated parameter of the dispatcher's Invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOutPoint has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

FieldOut is an alias for FieldOutPoint (since output arrays can only be defined on points).

FieldInOutPoint This tag represents field that is both an input and an output, which is necessarily associated with points. A FieldInOutPoint argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOutPoint has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

FieldInOut is an alias for FieldInOutPoint (since output arrays can only be defined on points).

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase, and this same object is given to the worklet. Execution objects are discussed in detail in Section 7.7 starting on page 125.

A field map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

CellCount This tag produces a vtkm::IdComponent equal to the number of cells incident on the point being visited. The Vecs provided from a FieldInCell parameter will be the same size as CellCount.

CellIndices This tag produces a Vec-like object of vtkm::Ids giving the indices for all incident cells. The order of the entries is arbitrary but will be consistent with the values of all other FieldInCell arguments for the same worklet invocation.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 7.8).

Cell to point field maps are typically used for converting fields associated with cells to points so that they can be interpolated. The following example does a simple averaging, but you can also implement other strategies such as a volume weighted average.

Example 7.8: Implementation and use of a map cell to point worklet.

```
#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/cont/DataSet.h>
#include <vtkm/cont/DataSetFieldAdd.h>
#include <vtkm/cont/DynamicArrayHandle.h>
#include <vtkm/cont/DynamicCellSet.h>
#include <vtkm/cont/Field.h>


class AverageCellField : public vtkm::worklet::WorkletMapCellToPoint
{
public:
  typedef void ControlSignature(TopologyIn cellSet,
                                FieldInCell<> inputCellField,
                                FieldOut<> outputPointField);
  typedef void ExecutionSignature(CellCount, _2, _3);
  typedef _1 InputDomain;

  template<typename InputCellFieldType, typename OutputFieldType>
  VTKM_EXEC_EXPORT
  void
  operator()(vtkm::IdComponent numCells,
             const InputCellFieldType &inputCellField,
             OutputFieldType &fieldAverage) const
  {
    // TODO: This trickery with calling DoAverage with an extra fabricated type
    // is to handle when the dynamic type resolution provides combinations that
    // are incompatible. On the todo list for VTK-m is to allow you to express
    // types that are the same for different parameters of the control
    // signature. When that happens, we can get rid of this hack.
    typedef typename InputCellFieldType::ComponentType InputComponentType;
    this->DoAverage(numCells,
                    inputCellField,
                    fieldAverage,
                    vtkm::ListTagBase<InputComponentType,OutputFieldType>());
  }

private:
  template<typename InputCellFieldType, typename OutputFieldType>
  VTKM_EXEC_EXPORT
  void DoAverage(vtkm::IdComponent numCells,
                 const InputCellFieldType &inputCellField,
                 OutputFieldType &fieldAverage,
                 vtkm::ListTagBase<OutputFieldType,OutputFieldType>) const
  {
    fieldAverage = OutputFieldType(0);

    for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
    {
      fieldAverage = fieldAverage + inputCellField[cellIndex];
    }

    fieldAverage = fieldAverage / OutputFieldType(numCells);
  }

  template<typename T1, typename T2, typename T3>
  VTKM_EXEC_EXPORT
  void DoAverage(vtkm::IdComponent, T1, T2, T3) const
  {
    this->RaiseError("Incompatible types for input and output.");
  }
};

VTKM_CONT_EXPORT
vtkm::cont::DataSet
ConvertCellFieldsToPointFields(const vtkm::cont::DataSet &inData)
{
  vtkm::cont::DataSet outData;

  // Copy parts of structure that should be passed through.
  for (vtkm::Id cellSetIndex = 0;
       cellSetIndex < inData.GetNumberOfCellSets();
       cellSetIndex++)
  {
    outData.AddCellSet(inData.GetCellSet(cellSetIndex));
  }
  for (vtkm::Id coordSysIndex = 0;
       coordSysIndex < inData.GetNumberOfCoordinateSystems();
       coordSysIndex++)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSysIndex));
  }
  // Copy all fields, converting cell fields to point fields.
  for (vtkm::Id fieldIndex = 0;
       fieldIndex < inData.GetNumberOfFields();
       fieldIndex++)
  {
    vtkm::cont::Field inField = inData.GetField(fieldIndex);
    if (inField.GetAssociation() == vtkm::cont::Field::ASSOC_CELL_SET)
    {
      vtkm::cont::DynamicArrayHandle inFieldData = inField.GetData();
      vtkm::cont::DynamicCellSet inCellSet =
          inData.GetCellSet(inField.GetAssocCellSet());

      vtkm::cont::DynamicArrayHandle outFieldData = inFieldData.NewInstance();
      vtkm::worklet::DispatcherMapTopology<AverageCellField> dispatcher;
      dispatcher.Invoke(inCellSet, inFieldData, outFieldData);

      vtkm::cont::DataSetFieldAdd::AddCellField(outData,
                                                inField.GetName(),
                                                outFieldData,
                                                inField.GetAssocCellSet());
    }
    else
    {
      outData.AddField(inField);
    }
  }

  return outData;
}
```

General Topology Maps

A worklet deriving vtkm::worklet::WorkletMapTopology performs a mapping operation that applies a function (the operator in the worklet) on all the elements of a specified type from a DataSet. While operating on each element, the worklet has access to fields associated both with that element and with all incident elements of a different specified type.

The WorkletMapTopology class is a template with two template parameters. The first template parameter specifies the "from" topology element, and the second template parameter specifies the "to" topology element. The worklet is scheduled such that each instance is associated with a particular "to" topology element and has access to incident "from" topology elements.

These from and to topology elements are specified with topology element tags, which are defined in the vtkm/TopologyElementTag.h header file. The available topology element tags are vtkm::TopologyElementTagCell, vtkm::TopologyElementTagPoint, vtkm::TopologyElementTagEdge, and vtkm::TopologyElementTagFace, which represent the cell, point, edge, and face elements, respectively.

WorkletMapTopology is a generic form of a topology map, and it can perform identically to the aforementioned forms of topology map with the correct template parameters. For example,

```
vtkm::worklet::WorkletMapTopology<vtkm::TopologyElementTagPoint,
    vtkm::TopologyElementTagCell>
```

is equivalent to the `vtkm::worklet::WorkletMapPointToCell` class except the signature tags have different names. The names used in the specific topology map superclasses (such as `WorkletMapPointToCell`) tend to be easier to read and are thus preferable. However, the generic `WorkletMapTopology` is available for topology combinations without a specific superclass or to support more general mappings in a worklet.

The general topology map worklet supports the following tags in the parameters of its `ControlSignature`, which are equivalent to tags in the other topology maps but with different (more general) names.

`TopologyIn` This tag represents the cell set that defines the collection of elements the map will operate on. A `TopologyIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Section 8.3.)

There must be exactly one `TopologyIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInFrom` This tag represents an input field that is associated with the "from" elements. A `FieldInFrom` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of "from" elements.

Each invocation of the worklet gets a Vec-like object containing the field values for all the "from" elements incident with the "to" element being visited. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInFrom` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldInTo` This tag represents an input field that is associated with the "to" element. A `FieldInTo` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

`FieldInTo` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

`FieldOut` This tag represents an output field, which is necessarily associated with "to" elements. A `FieldOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

FieldInOut This tag represents field that is both an input and an output, which is necessarily associated with "to" elements. A FieldInOut argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the dispatcher's Invoke. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 7.6 starting on page 122.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 7.4.1 starting on page 104.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase, and this same object is given to the worklet. Execution objects are discussed in detail in Section 7.7 starting on page 125.

A general topology map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the `ControlSignature`.

`CellShape` This tag produces a shape tag corresponding to the shape of the visited "to" element. (Cell shapes and the operations you can do with cells are discussed in Section 8.3.) This is the same value that gets provided if you reference the `TopologyIn` parameter.

If the "to" element is cells, the `CellShape` clearly will match the shape of each cell. Other elements will have shapes to match their structures. Points have vertex shapes, edges have line shapes, and faces have some type of polygonal shape.

`FromCount` This tag produces a `vtkm::IdComponent` equal to the number of "from" elements incident on the "to" element being visited. The Vecs provided from a `FieldInFrom` parameter will be the same size as `FromCount`.

`FromIndices` This tag produces a Vec-like object of `vtkm::Id`s giving the indices for all incident "from" elements. The order of the entries is consistent with the values of all other `FieldInFrom` arguments for the same worklet invocation.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 7.8).

7.6 Whole Arrays

As documented in Section 7.5, each worklet type has a set of parameter types that can be used to pass data to and from the worklet invocation. But what happens if you want to pass data that cannot be expressed in these predefined mechanisms. Chapter 9 describes how to create completely new worklet types and parameter tags. However, designing such a system for a one-time use is overkill.

Instead, all VTK-m worklets provide a couple of mechanisms that allow you to pass arbitrary data to a worklet. In this section, we will explore a *whole array* argument that provides random access to an entire array. In the following section we describe an even more general mechanism to describe any execution object.

We have already seen a demonstration of using a whole array in Example 7.6 to perform a simple array copy. Here we will construct a more thorough example of building functionality that requires random array access.

Let's say we want to measure the quality of triangles in a mesh. A common method for doing this is using the equation

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where a is the area of the triangle and h_1, h_2, and h_3 are the lengths of the sides. We can easily compute this in a cell to point map, but what if we want to speed up the computations by reducing precision? After all, we probably only care if the triangle is good, reasonable, or bad. So instead, let's build a lookup table and then retrieve the triangle quality from that lookup table based on its sides.

The following example demonstrates creating such a table lookup in an array and using a worklet argument tagged with `WholeArrayIn` to make it accessible.

Example 7.9: Using `WholeArrayIn` to access a lookup table in a worklet.

```
#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/DataSet.h>

#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/CellShape.h>
#include <vtkm/Math.h>
#include <vtkm/VectorAnalysis.h>

static const vtkm::Id TRIANGLE_QUALITY_TABLE_DIMENSION = 8;
static const vtkm::Id TRIANGLE_QUALITY_TABLE_SIZE =
    TRIANGLE_QUALITY_TABLE_DIMENSION*TRIANGLE_QUALITY_TABLE_DIMENSION;

VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32> GetTriangleQualityTable()
{
  // Use these precomputed values for the array. A real application would
  // probably use a larger array, but we are keeping it small for demonstration
  // purposes.
  static vtkm::Float32 triangleQualityBuffer[TRIANGLE_QUALITY_TABLE_SIZE] = {
    0, 0,        0,        0,        0,        0,        0,        0,
    0, 0,        0,        0,        0,        0,        0,        0.244311,
    0, 0,        0,        0,        0,        0,        0.432985, 0.470588,
    0, 0,        0,        0,        0,        0.542168, 0.659231, 0.664078,
    0, 0,        0,        0,        0.579721, 0.754247, 0.821543, 0.815365,
    0, 0,        0,        0.542168, 0.754247, 0.874598, 0.925667, 0.920712,
    0, 0,        0.432985, 0.659231, 0.821543, 0.925667, 0.976641, 0.980996,
    0, 0.244311, 0.470588, 0.664078, 0.815365, 0.920712, 0.980996, 1
  };

  return vtkm::cont::make_ArrayHandle(triangleQualityBuffer,
                                      TRIANGLE_QUALITY_TABLE_SIZE);
}

template<typename T>
VTKM_EXEC_CONT_EXPORT
vtkm::Vec<T,3> TriangleEdgeLengths(const vtkm::Vec<T,3> &point1,
                                   const vtkm::Vec<T,3> &point2,
                                   const vtkm::Vec<T,3> &point3)
{
  return vtkm::make_Vec(vtkm::Magnitude(point1-point2),
                        vtkm::Magnitude(point2-point3),
                        vtkm::Magnitude(point3-point1));
}

VTKM_SUPPRESS_EXEC_WARNINGS
```

```
template<typename PortalType, typename T>
VTKM_EXEC_CONT_EXPORT
vtkm::Float32 LookupTriangleQuality(const PortalType &triangleQualityPortal,
                                    const vtkm::Vec<T,3> &point1,
                                    const vtkm::Vec<T,3> &point2,
                                    const vtkm::Vec<T,3> &point3)
{
  vtkm::Vec<T,3> edgeLengths = TriangleEdgeLengths(point1, point2, point3);

  // To reduce the size of the table, we just store the quality of triangles
  // with the longest edge of size 1. The table is 2D indexed by the length
  // of the other two edges. Thus, to use the table we have to identify the
  // longest edge and scale appropriately.
  T smallEdge1 = vtkm::Min(edgeLengths[0], edgeLengths[1]);
  T tmpEdge = vtkm::Max(edgeLengths[0], edgeLengths[1]);
  T smallEdge2 = vtkm::Min(edgeLengths[2], tmpEdge);
  T largeEdge = vtkm::Max(edgeLengths[2], tmpEdge);

  smallEdge1 /= largeEdge;
  smallEdge2 /= largeEdge;

  // Find index into array.
  vtkm::Id index1 = static_cast<vtkm::Id>(
        vtkm::Floor(smallEdge1*(TRIANGLE_QUALITY_TABLE_DIMENSION-1)+0.5));
  vtkm::Id index2 = static_cast<vtkm::Id>(
        vtkm::Floor(smallEdge2*(TRIANGLE_QUALITY_TABLE_DIMENSION-1)+0.5));
  vtkm::Id totalIndex = index1 + index2*TRIANGLE_QUALITY_TABLE_DIMENSION;

  return triangleQualityPortal.Get(totalIndex);
}

struct TriangleQualityWorklet : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(TopologyIn cells,
                                FieldInPoint<Vec3> pointCoordinates,
                                WholeArrayIn<Scalar> triangleQualityTable,
                                FieldOutCell<Scalar> triangleQuality);
  typedef _4 ExecutionSignature(CellShape, _2, _3);
  typedef _1 InputDomain;

  template<typename CellShape,
           typename PointCoordinatesType,
           typename TriangleQualityTablePortalType>
  VTKM_EXEC_EXPORT
  vtkm::Float32 operator()(
      CellShape shape,
      const PointCoordinatesType &pointCoordinates,
      const TriangleQualityTablePortalType &triangleQualityTable) const
  {
    if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
    {
      this->RaiseError("Only triangles are supported for triangle quality.");
      return vtkm::Nan32();
    }

    return LookupTriangleQuality(triangleQualityTable,
                                 pointCoordinates[0],
                                 pointCoordinates[1],
                                 pointCoordinates[2]);
  }
};

// Normally we would encapsulate this call in a filter, but for demonstrative
// purposes we are just calling the worklet directly.
template<typename DeviceAdapterTag>
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32>
RunTriangleQuality(vtkm::cont::DataSet dataSet,
                   DeviceAdapterTag)
{
  vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualityTable =
      GetTriangleQualityTable();

  vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;

  vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet,DeviceAdapterTag>
      dispatcher;
  dispatcher.Invoke(dataSet.GetCellSet(),
                    dataSet.GetCoordinateSystem().GetData(),
                    triangleQualityTable,
                    triangleQualities);

  return triangleQualities;
}
```

7.7 Execution Objects

Although passing whole arrays into a worklet is a convenient way to provide data to a worklet that is not divided by the input or output domain, it is sometimes not the best structure to represent data. Thus, all worklets support a second type of argument called an *execution object*, or exec object for short, that passes the given object directly to each invocation of the worklet. This is defined by an `ExecObject` tag in the `ControlSignature`.

The execution object must be a subclass of `vtkm::exec::ExecutionObjectBase`. Also, it must be possible to copy the object from the control environment to the execution environment and be usable in the execution environment, and any method of the execution object used within the worklet must be declared with `VTKM_EXEC_EXPORT` or `VTKM_EXEC_CONT_EXPORT`.

An execution object can refer to an array, but the array reference must be through an array portal for the execution environment. This can be retrieved from the `PrepareForInput` method in `vtkm::cont::ArrayHandle` as described in Section 6.2.3. Other VTK-m data objects, such as the subclasses of `vtkm::cont::CellSet`, have similar methods.

Returning to the example we have in Section 7.6, we are computing triangle quality quickly by looking up the value in a table. In Example 7.9 the table is passed directly to the worklet as a whole array. However, there is some additional code involved to get the appropriate index into the table for a given triangle. Let us say that we want to have the ability to compute triangle quality in many different worklets. Rather than pass in a raw array, it would be better to encapsulate the functionality in an object.

We can do that by creating an execution object that has the table stored inside and methods to compute the triangle quality. The following example uses the table built in Example 7.9 to create such an object.

Example 7.10: Using `ExecObject` to access a lookup table in a worklet.

```
template<typename DeviceAdapterTag>
```

```
class TriangleQualityTable : public vtkm::exec::ExecutionObjectBase
{
public:
  VTKM_CONT_EXPORT
  TriangleQualityTable()
  {
    this->TablePortal =
        GetTriangleQualityTable().PrepareForInput(DeviceAdapterTag());
  }

  template<typename T>
  VTKM_EXEC_EXPORT
  vtkm::Float32 GetQuality(const vtkm::Vec<T,3> &point1,
                           const vtkm::Vec<T,3> &point2,
                           const vtkm::Vec<T,3> &point3) const
  {
    return LookupTriangleQuality(this->TablePortal, point1, point2, point3);
  }

private:
  typedef vtkm::cont::ArrayHandle<vtkm::Float32> TableArrayType;
  typedef typename TableArrayType::ExecutionTypes<DeviceAdapterTag>::PortalConst
      TableArrayPortalType;
  TableArrayPortalType TablePortal;
};

struct TriangleQualityWorklet2 : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(TopologyIn cells,
                                FieldInPoint<Vec3> pointCoordinates,
                                ExecObject triangleQualityTable,
                                FieldOutCell<Scalar> triangleQuality);
  typedef _4 ExecutionSignature(CellShape, _2, _3);
  typedef _1 InputDomain;

  template<typename CellShape,
           typename PointCoordinatesType,
           typename TriangleQualityTableType>
  VTKM_EXEC_EXPORT
  vtkm::Float32 operator()(
      CellShape shape,
      const PointCoordinatesType &pointCoordinates,
      const TriangleQualityTableType &triangleQualityTable) const
  {
    if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
    {
      this->RaiseError("Only triangles are supported for triangle quality.");
      return vtkm::Nan32();
    }

    return triangleQualityTable.GetQuality(pointCoordinates[0],
                                           pointCoordinates[1],
                                           pointCoordinates[2]);
  }
};

// Normally we would encapsulate this call in a filter, but for demonstrative
// purposes we are just calling the worklet directly.
template<typename DeviceAdapterTag>
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32>
RunTriangleQuality2(vtkm::cont::DataSet dataSet,
                    DeviceAdapterTag)
{
  TriangleQualityTable<DeviceAdapterTag> triangleQualityTable;

  vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
  vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet2,DeviceAdapterTag>
      dispatcher;
  dispatcher.Invoke(dataSet.GetCellSet(),
                    dataSet.GetCoordinateSystem().GetData(),
                    triangleQualityTable,
                    triangleQualities);

  return triangleQualities;
}
```

7.8 Scatter

The default scheduling of a worklet provides a 1 to 1 mapping from the input domain to the output domain. For example, a `vtkm::worklet::WorkletMapField` gets run once for every item of the input array and produces one item for the output array. Likewise, `vtkm::worklet::WorkletMapPointToCell` gets run once for every cell in the input topology and produces one associated item for the output field.

However, there are many operations that do not fall well into this 1 to 1 mapping procedure. The operation might need to pass over elements that produce no value or the operation might need to produce multiple values for a single input element.

Such non 1 to 1 mappings can be achieved by defining a *scatter* for a worklet. The following types of scatter are provided by VTK-m.

`vtkm::worklet::ScatterIdentity` Provides a basic 1 to 1 mapping from input to output. This is the default scatter used if none is specified.

`vtkm::worklet::ScatterUniform` Provides a 1 to many mapping from input to output with the same number of outputs for each input. The worklet provides a number at runtime that defines the number of output values to produce per input.

`vtkm::worklet::ScatterCounting` Provides a 1 to any mapping from input to output with different numbers of outputs for each input. The worklet provides an `ArrayHandle` that is the same size as the input containing the count of output values to produce for each input. Values can be zero, in which case that input will be skipped.

To define a scatter procedure, the worklet must provide two items. The first item is a `typedef` named `ScatterType`. The `ScatterType` must be `typedef`ed to one of the aforementioned `Scatter*` classes. The second item is a `const` method named `GetScatter` that returns an object of type `ScatterType`.

Example 7.11: Declaration of a scatter type in a worklet.

```
  typedef vtkm::worklet::ScatterCounting ScatterType;

  VTKM_CONT_EXPORT
  ScatterType GetScatter() const { return this->Scatter; }
```

When using a scatter that produces multiple outputs for a single input, the worklet is invoked multiple times with the same input values. In such an event the worklet operator needs to distinguish these calls to produce the correct associated output. This is done by declaring one of the `ExecutionSignature` arguments as `VisitIndex`. This tag will pass a `vtkm::IdComponent` to the worklet that identifies which invocation is being called.

To demonstrate using scatters with worklets, we provide some contrived but illustrative examples. The first example is a worklet that takes a pair of input arrays and interleaves them so that the first, third, fifth, and so on entries come from the first array and the second, fourth, sixth, and so on entries come from the second array. We achieve this by using a `vtkm::cont::ScatterUniform` of size 2 and using the `VisitIndex` to determine from which array to pull a value.

Example 7.12: Using `ScatterUniform`.

```
struct InterleaveArrays : vtkm::worklet::WorkletMapField
{
  typedef void ControlSignature(FieldIn<>, FieldIn<>, FieldOut<>);
  typedef void ExecutionSignature(_1, _2, _3, VisitIndex);
  typedef _1 InputDomain;

  typedef vtkm::worklet::ScatterUniform ScatterType;

  VTKM_CONT_EXPORT
  ScatterType GetScatter() const { return vtkm::worklet::ScatterUniform(2); }

  template<typename T>
  VTKM_EXEC_EXPORT
  void operator()(const T &input0,
                  const T &input1,
                  T &output,
                  vtkm::IdComponent visitIndex) const
  {
    if (visitIndex == 0)
    {
      output = input0;
    }
    else // visitIndex == 1
    {
      output = input1;
    }
  }
};
```

The second example takes a collection of point coordinates and clips them by an axis-aligned bounding box. It does this using a `vtkm::cont::ScatterCounting` with an array containing 0 for all points outside the bounds and 1 for all points inside the bounds. As is typical with this type of operation, we use another worklet with a default identity scatter to build the count array.

Example 7.13: Using `ScatterCounting`.

```
class ClipPointsCount : public vtkm::worklet::WorkletMapField
{
public:
  typedef void ControlSignature(FieldIn<Vec3> points,
                                FieldOut<IdComponentType> count);
  typedef _2 ExecutionSignature(_1);
  typedef _1 InputDomain;
  template<typename T>
  VTKM_CONT_EXPORT
  ClipPointsCount(const vtkm::Vec<T,3> &boundsMin,
                  const vtkm::Vec<T,3> &boundsMax)
    : BoundsMin(boundsMin[0], boundsMin[1], boundsMin[2]),
      BoundsMax(boundsMax[0], boundsMax[1], boundsMax[2])
  {  }

  template<typename T>
  VTKM_EXEC_EXPORT
  vtkm::IdComponent operator()(const vtkm::Vec<T,3> &point) const
  {
    return static_cast<vtkm::IdComponent>((this->BoundsMin[0] < point[0]) &&
                                          (this->BoundsMin[1] < point[1]) &&
                                          (this->BoundsMin[2] < point[2]) &&
                                          (this->BoundsMax[0] > point[0]) &&
                                          (this->BoundsMax[1] > point[1]) &&
                                          (this->BoundsMax[2] > point[2]));
  }

private:
  vtkm::Vec<vtkm::FloatDefault,3> BoundsMin;
  vtkm::Vec<vtkm::FloatDefault,3> BoundsMax;
};

class ClipPointsGenerate : public vtkm::worklet::WorkletMapField
{
public:
  typedef void ControlSignature(FieldIn<Vec3> inPoints,
                                FieldOut<Vec3> outPoints);
  typedef void ExecutionSignature(_1, _2);
  typedef _1 InputDomain;

  typedef vtkm::worklet::ScatterCounting ScatterType;

  VTKM_CONT_EXPORT
  ScatterType GetScatter() const { return this->Scatter; }

  template<typename CountArrayType, typename DeviceAdapterTag>
  VTKM_CONT_EXPORT
  ClipPointsGenerate(const CountArrayType &countArray, DeviceAdapterTag)
    : Scatter(countArray, DeviceAdapterTag())
  {
    VTKM_IS_ARRAY_HANDLE(CountArrayType);
  }

  template<typename InType, typename OutType>
  VTKM_EXEC_EXPORT
  void operator()(const vtkm::Vec<InType,3> &inPoint,
                  vtkm::Vec<OutType,3> &outPoint) const
  {
    // The scatter ensures that this method is only called for input points
    // that are passed to the output (where the count was 1). Thus, in this
    // case we know that we just need to copy the input to the output.
    outPoint = vtkm::Vec<OutType,3>(inPoint[0], inPoint[1], inPoint[2]);
  }

private:
  ScatterType Scatter;
};

// Normally we would encapsulate these calls in a filter, but for demonstrative
// purposes we are just calling the worklet directly.
template<typename T, typename Storage, typename DeviceAdapterTag>
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Vec<T,3> >
RunClipPoints(const vtkm::cont::ArrayHandle<vtkm::Vec<T,3>, Storage> &pointArray,
               vtkm::Vec<T,3> boundsMin,
               vtkm::Vec<T,3> boundsMax,
               DeviceAdapterTag)
{
  vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray;

  ClipPointsCount workletCount(boundsMin, boundsMax);
  vtkm::worklet::DispatcherMapField<ClipPointsCount, DeviceAdapterTag>
      dispatcherCount(workletCount);
  dispatcherCount.Invoke(pointArray, countArray);

  vtkm::cont::ArrayHandle<vtkm::Vec<T,3> > clippedPointsArray;

  ClipPointsGenerate workletGenerate(countArray, DeviceAdapterTag());
  vtkm::worklet::DispatcherMapField<ClipPointsGenerate, DeviceAdapterTag>
      dispatcherGenerate(workletGenerate);
  dispatcherGenerate.Invoke(pointArray, clippedPointsArray);

  return clippedPointsArray;
}
```

Chapter 8

Execution Environment

The execution environment is exposed to developers that write worklets for different visualization algorithms. In addition to providing all the mechanisms for building the worklet object itself, the execution environment contains supporting code that can be useful to the implementations of visualization algorithms.

The data structures in the execution environment provide information and operations for a single element. This is in contrast to the control environment, where data structures are built on arrays providing information for large collections of data. These respective data structures reflect the nature of the two environments. The control environment manages the stores of data whereas the execution environment performs large parallel processing through fine operations.

8.1 Error Handling

It is sometimes the case during the execution of an algorithm that an error condition can occur that causes the computation to become invalid. At such a time, it is important to raise an error to alert the calling code of the problem. Since VTK-m uses an exception mechanism to raise errors, we want an error in the execution environment to throw an exception.

However, throwing exceptions in a parallel algorithm is problematic. Some accelerator architectures, like CUDA, do not even support throwing exceptions. Even on architectures that do support exceptions, throwing them in a thread block can cause problems. An exception raised in one thread may or may not be thrown in another, which increases the potential for deadlocks, and it is unclear how uncaught exceptions progress through thread blocks.

VTK-m handles this problem by using a flag and check mechanism. When a worklet (or other subclass of `vtkm::exec::FunctorBase`) encounters an error, it can call its `RaiseError` method to flag the problem and record a message for the error. Once all the threads terminate, the scheduler checks for the error, and if one exists it throws a `vtkm::cont::ErrorExecution` exception in the control environment. Thus, calling `RaiseError` looks like an exception was thrown from the perspective of the control environment code that invoked it.

Example 8.1: Raising an error in the execution environment.

```
struct SquareRoot : vtkm::worklet::WorkletMapField
{
public:
  typedef void ControlSignature(FieldIn<Scalar>, FieldOut<Scalar>);
  typedef _2 ExecutionSignature(_1);

  template<typename T>
  VTKM_EXEC_EXPORT
  T operator()(T x) const
  {
    if (x < 0)
      {
      this->RaiseError("Cannot take the square root of a negative number.");
      }
    return vtkm::Sqrt(x);
  }
};
```

8.2 Math

VTK-m comes with several math functions that tend to be useful for visualizatio algorithms. The implementation of basic math operations can vary subtly on different accelerators, and these functions provide cross platform support.

All math functions are located in the `vtkm` package. The functions are most useful in the execution environment (and are hence documented in this chapter), but they can also be used in the control environment when needed.

8.2.1 Basic Math

The vtkm/Math.h header file contains several math functions that replicate the behavior of the basic POSIX math functions as well as related functionality.

`vtkm::Abs` Returns the absolute value of the single argument. If given a vector, performs a component-wise operation.

`vtkm::ACos` Returns the arccosine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ACosH` Returns the hyperbolic arccossine. If given a vector, performs a component-wise operation.

`vtkm::ASin` Returns the arcsine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ASinH` Returns the hyperbolic arcsine. If given a vector, performs a component-wise operation.

vtkm::ATan Returns the arctangent of a ratio in radians. If given a vector, performs a component-wise operation.

vtkm::ATan2 Computes the arctangent of y/x where y is the first argument and x is the second argument. `ATan2` uses the signs of both arguments to determine the quadrant of the return value. `ATan2` is only defined for floating point types (no vectors).

vtkm::ATanH Returns the hyperbolic arctangent. If given a vector, performs a component-wise operation.

vtkm::Cbrt Takes one argument and returns the cube root of that argument. If called with a vector type, returns a component-wise cube root.

vtkm::Ceil Rounds and returns the smallest integer not less than the single argument. If given a vector, performs a component-wise operation.

vtkm::CopySign Copies the sign of the second argument onto the first argument and returns that. If the second argument is positive, returns the absolute value of the first argument. If the second argument is negative, returns the negative absolute value of the first argument.

vtkm::Cos Returns the cosine of an angle given in radians. If given a vector, performs a component-wise operation.

vtkm::CosH Returns the hyperbolic cosine. If given a vector, performs a component-wise operation.

vtkm::Epsilon Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Epsilon32` and `Epsilon64` are non-templated versions that return the precision for a particular precision.

vtkm::Exp Computes e^x where x is the argument to the function and e is Euler's number (approximately 2.71828). If called with a vector type, returns a component-wise exponent.

vtkm::Exp10 Computes 10^x where x is the argument. If called with a vector type, returns a component-wise exponent.

vtkm::Exp2 Computes 2^x where x is the argument. If called with a vector type, returns a component-wise exponent.

vtkm::ExpM1 Computes $e^x - 1$ where x is the argument to the function and e is Euler's number (approximately 2.71828). The accuracy of this function is good even for very small values of x. If called with a vector type, returns a component-wise exponent.

vtkm::Floor Rounds and returns the largest integer not greater than the single argument. If given a vector, performs a component-wise operation.

vtkm::FMod Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards zero to an integer. For example, `FMod(6.5,2.3)` returns 1.9, which is $6.5 - 2 \cdot 4.6$. If given vectors, `FMod` performs a component-wise operation. `FMod` is similar to `Remainder` except that the quotient is rounded toward 0 instead of the nearest integer.

vtkm::Infinity Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Infinity32` and `Infinity64` are non-templated versions that return the precision for a particular precision.

vtkm::IsFinite Returns true if the argument is a normal number (neither a NaN nor an infinite).

vtkm::IsInf Returns true if the argument is either positive infinity or negative infinity.

vtkm::IsNan Returns true if the argument is not a number (NaN).

vtkm::IsNegative Returns true if the single argument is less than zero, false otherwise.

vtkm::Log Computes the natural logarithm (i.e. logarithm to the base e) of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log10 Computes the logarithm to the base 10 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log1P Computes $\ln(1 + x)$ where x is the single argument and ln is the natural logarithm (i.e. logarithm to the base e). The accuracy of this function is good for very small values. If called with a vector type, returns a component-wise logarithm.

vtkm::Log2 Computes the logarithm to the base 2 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Max Takes two arguments and returns the argument that is greater. If called with a vector type, returns a component-wise maximum.

vtkm::Min Takes two arguments and returns the argument that is lesser. If called with a vector type, returns a component-wise minimum.

vtkm::ModF Returns the integral and fractional parts of the first argument. The second argument is a reference in which the integral part is stored. The return value is the fractional part. If given vectors, `ModF` performs a component-wise operation.

`vtkm::Nan` Returns the representation for not-a-number (NaN). A NaN represents an invalid value or the result of an invalid operation such as 0/0. A NaN is neither greater than nor less than nor equal to any other number including other NaNs. The `NaN` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Nan32` and `NaN64` are non-templated versions that return the precision for a particular precision.

`vtkm::NegativeInfinity` Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The `NegativeInfinity` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `NagativeInfinity32` and `NegativeInfinity64` are non-templated versions that return the precision for a particular precision.

`vtkm::Pi` Returns the constant π (about 3.14159).

`vtkm::Pi_2` Returns the constant $\pi/2$ (about 1.570796).

`vtkm::Pi_3` Returns the constant $\pi/3$ (about 1.047197).

`vtkm::Pi_4` Returns the constant $\pi/4$ (about 0.785398).

`vtkm::Pow` Takes two arguments and returns the first argument raised to the power of the second argument. This function is only defined for `vtkm::Float32` and `vtkm::Float64`.

`vtkm::RCbrt` Takes one argument and returns the cube root of that argument. The result of this function is equivalent to `1/Cbrt(x)`. However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

`vtkm::Remainder` Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer. For example, `FMod(6.5,2.3)` returns -0.4, which is $6.5 - 3 \cdot 2.3$. If given vectors, `Remainder` performs a component-wise operation. `Remainder` is similar to `FMod` except that the quotient is rounded toward the nearest integer instead of toward 0.

`vtkm::RemainderQuotient` Performs an operation identical to `Reminder`. In addition, this function takes a third argument that is a reference in which the quotient is given.

`vtkm::Round` Rounds and returns the integer nearest the single argument. If given a vector, performs a component-wise operation.

`vtkm::RSqrt` Takes one argument and returns the square root of that argument. The result of this function is equivalent to `1/Sqrt(x)`. However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

`vtkm::SignBit` Returns a nonzero value if the single argument is negative.

`vtkm::Sin` Returns the sine of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::SinH` Returns the hyperbolic sine. If given a vector, performs a component-wise operation.

`vtkm::Sqrt` Takes one argument and returns the square root of that argument. If called with a vector type, returns a component-wise square root. On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide byt the square root.

`vtkm::Tan` Returns the tangent of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::TanH` Returns the hyperbolic tangent. If given a vector, performs a component-wise operation.

`vtkm::TwoPi` Returns the constant 2π (about 6.283185).

8.2.2 Vector Analysis

Visualization and computational geometry algorithms often perform vector analysis operations. The vtkm/VectorAnalysis.h header file provides functions that perform the basic common vector analysis operations.

`vtkm::Cross` Returns the cross product of two `vtkm::Vec` of size 3.

`vtkm::Lerp` Given two values x and y in the first two parameters and a weight w as the third parameter, interpolates between x and y. Specifically, the linear interpolation is $(y - x)w + x$ although `Lerp` might compute the interpolation faster than using the independent arithmetic operations. The two values may be scalars or equal sized vectors. If the two values are vectors and the weight is a scalar, all components of the vector are interpolated with the same weight. If the weight is also a vector, then each component of the value vectors are interpolated with the respective weight component.

`vtkm::Magnitude` Returns the magnitude of a vector. This function works on scalars as well as vectors, in which case it just returns the scalar. It is usually much faster to compute `MagnitudeSquared`, so that should be substituted when possible (unless you are just going to take the square root, which would be besides the point). On some hardware it is also faster to find the reciprocal magnitude, so `RMagnitude` should be used if you actually plan to divide by the magnitude.

vtkm::MagnitudeSquared Returns the square of the magnitude of a vector. It is usually much faster to compute the square of the magnitude than the length, so you should use this function in place of `Magnitude` or `RMagnitude` when needing the square of the magnitude or any monotonically increasing function of a magnitude or distance. This function works on scalars as well as vectors, in which case it just returns the square of the scalar.

vtkm::Normal Returns a normalized version of the given vector. The resulting vector points in the same direction as the argument but has unit length.

vtkm::Normalize Takes a reference to a vector and modifies it to be of unit length. `Normalize(v)` is functionally equivalent to `v *= RMagnitude(v)`.

vtkm::RMagnitude Returns the reciprocal magnitude of a vector. On some hardware `RMagnitude` is faster than `Magnitude`, but neither is as fast as `MagnitudeSquared`. This function works on scalars as well as vectors, in which case it just returns the reciprocal of the scalar.

vtkm::TriangleNormal Given three points in space (contained in `vtkm::Vec`s of size 3) that compose a triangle return a vector that is perpendicular to the triangle. The magnitude of the result is equal to twice the area of the triangle. The result points away from the "front" of the triangle as defined by the standard counter-clockwise ordering of the points.

8.2.3 Matrices

Linear algebra operations on small matrices that are done on a single thread are located in dax/Matrix.h.

This header defines the `vtkm::Matrix` templated class. The template parameters are first the type of component, then the number of rows, then the number of columns. The overloaded parentheses operator can be used to retrieve values based on row and column indices. Likewise, the bracket operators can be used to reference the `Matrix` as a 2D array (indexed by row first). The following example builds a `Matrix` that contains the values

$$\begin{vmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \end{vmatrix}$$

Example 8.2: Creating a `Matrix`.

```
vtkm::Matrix<vtkm::Float32, 2, 3> matrix;

// Using parenthesis notation.
matrix(0,0) = 0.0f;
matrix(0,1) = 1.0f;
matrix(0,2) = 2.0f;
```

```
// Using bracket notation.
matrix[1][0] = 10.0f;
matrix[1][1] = 11.0f;
matrix[1][2] = 12.0f;
```

The dax/Matrix.h header also defines the following functions that operate on matrices.

vtkm::MatrixDeterminant Takes a square Matrix as its single argument and returns the determinant of that matrix.

vtkm::MatrixGetColumn Given a Matrix and a column index, returns a vtkm::Vec of that column. This function might not be as efficient as vtkm::MatrixRow. (It performs a copy of the column).

vtkm::MatrixGetRow Given a Matrix and a row index, returns a vtkm::Vec of that row.

vtkm::MatrixIdentity Returns the identity matrix. If given no arguments, it creates an identity matrix and returns it. (In this form, the component type and size must be explicitly set.) If given a single square matrix argument, fills that matrix with the identity.

vtkm::MatrixInverse Finds and returns the inverse of a given matrix. The function takes two arguments. The first argument is the matrix to invert. The second argument is a reference to a Boolean that is set to true if the inverse is found or false if the matrix is singular and the returned matrix is incorrect.

vtkm::MatrixMultiply Performs a matrix-multiply on its two arguments. Overloaded to work for matrix-matrix, vector-matrix, or matrix-vector multiply.

vtkm::MatrixSetColumn Given a Matrix, a column index, and a vtkm::Vec, sets the column of that index to the values of the Tuple.

vtkm::MatrixSetRow Given a Matrix, a row index, and a vtkm::Vec, sets the row of that index to the values of the Tuple.

vtkm::MatrixTranspose Takes a Matrix and returns its transpose.

vtkm::SolveLinearSystem Solves the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and returns $\boldsymbol{x}$. The function takes three arguments. The first two arguments are the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$, respectively. The third argument is a reference to a Boolean that is set to true if a single solution is found, false otherwise.

8.2.4 Newton's Method

VTK-m's matrix methods (documented in Section 8.2.3) provide a method to solve a small linear system of equations. However, sometimes it is necessary to solve a small nonlinear system of equations. This can be done with the vtkm::exec::NewtonsMethod function defined in the vtkm/exec/NewtonsMethod header.

[NewtonsMethod is defined in the execution environment because of its call-back functions. Since it was created, the VTKM_SUPPRESS_EXEC_WARNINGS has been introduced to handle that. We should move this method to the base vtkm environment.]

The NewtonsMethod function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the NewtonsMethod. The function takes the following 6 parameters (three of which are optional).

1. A functor whose operation takes a vtkm::Vec and returns a vtkm::Matrix containing the math function's Jacobian vector at that point.
2. A functor whose operation takes a vtkm::Vec and returns the evaluation of the math function at that point as another vtkm::Vec.
3. The vtkm::Vec that represents the desired output of the function.
4. A vtkm::Vec to use as the initial guess. If not specified, the origin is used.
5. The convergence distance. If the iterative method changes all values less than this amount, then it considers the solution found. If not specified, set to 10^{-3}.
6. The maximum amount of iterations to run before giving up and returning the best solution. If not specified, set to 10.

Example 8.3: Using NewtonsMethod to solve a small system of nonlinear equations.

```
// A functor for the mathematical function f(x) = [dot(x,x),x[0]*x[1]]
struct FunctionFunctor
{
  template<typename T>
  VTKM_EXEC_CONT_EXPORT
  vtkm::Vec<T,2> operator()(const vtkm::Vec<T,2> &x) const
  {
    return vtkm::make_Vec(vtkm::dot(x,x), x[0]*x[1]);
  }
};

// A functor for the Jacobian of the mathematical function
// f(x) = [dot(x,x),x[0]*x[1]], which is
//   | 2*x[0] 2*x[1] |
//   |   x[1]   x[0] |
struct JacobianFunctor
{
  template<typename T>
  VTKM_EXEC_CONT_EXPORT
  vtkm::Matrix<T,2,2> operator()(const vtkm::Vec<T,2> &x) const
  {
    vtkm::Matrix<T,2,2> jacobian;
    jacobian(0,0) = 2*x[0];
    jacobian(0,1) = 2*x[1];
    jacobian(1,0) = x[1];
    jacobian(1,1) = x[0];

    return jacobian;
  }
};

VTKM_EXEC_EXPORT
void SolveNonlinear()
{
  // Use Newton's method to solve the nonlinear system of equations:
  //
  //     x^2 + y^2 = 2
  //     x*y = 1
  //
  // There are two possible solutions, which are (x=1,y=1) and (x=-1,y=-1).
  // The one found depends on the starting value.
  vtkm::Vec<vtkm::Float32,2> answer1 =
      vtkm::exec::NewtonsMethod(JacobianFunctor(),
                                FunctionFunctor(),
                                vtkm::make_Vec(2.0f, 1.0f),
                                vtkm::make_Vec(1.0f, 0.0f));
  // answer1 is [1,1]

  vtkm::Vec<vtkm::Float32,2> answer2 =
      vtkm::exec::NewtonsMethod(JacobianFunctor(),
                                FunctionFunctor(),
                                vtkm::make_Vec(2.0f, 1.0f),
                                vtkm::make_Vec(0.0f, -2.0f));
  // answer2 is [-1,-1]
}
```

8.3 Working with Topology

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See Section 6.4.2 starting on page 81 for information on defining cell sets in the control environment.) When worklets that operate on cells are scheduled, these grid structures are broken into their independent cells, and that data is handed to the worklet. Thus, cell-based operations in the execution environment exclusively operate on independent cells.

Unlike some other libraries such as VTK, VTK-m does not have a cell class that holds all the information pertaining to a cell of a particular type. Instead, VTK-m provides tags or identifiers defining the cell shape, and companion data like coordinate and field information are held in separate structures. This organization is designed so a worklet may specify exactly what information it needs, and only that information will be loaded.

8.3.1 Cell Shape Tags and Ids

Cell shapes can be specified with either a tag (defined with a struct with a name like `CellShapeTag*`) or an enumerated identifier (defined with a constant number with a name like `CELL_SHAPE_*`). These shape tags and identifiers are defined in vtkm/CellShape.h and declared in the `vtkm` namespace (because they can be used in either the control or the execution

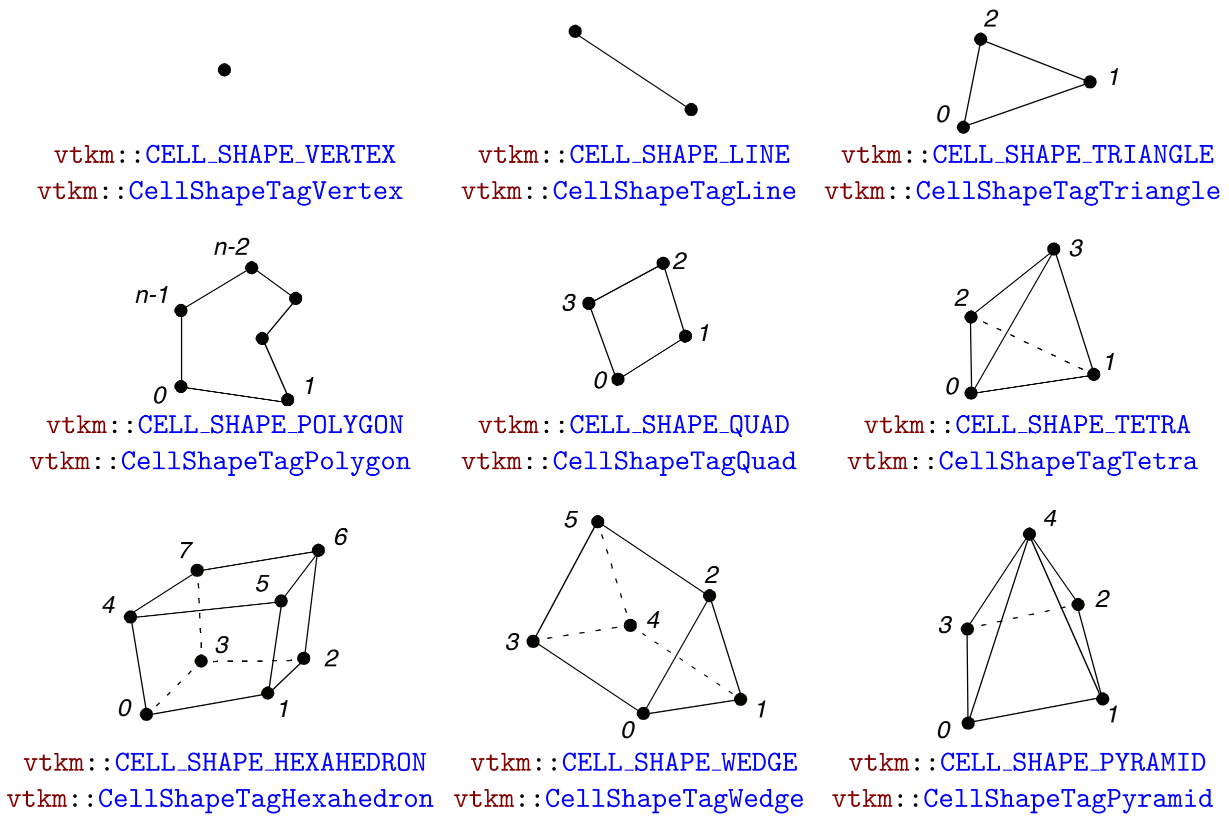


Figure 8.1: Basic Cell Shapes

environment). Figure 8.1 gives both the identifier and the tag names.

In addition to the basic cell shapes, there is a special "empty" cell with the identifier vtkm::CELL_SHAPE_EMPTY and tag vtkm::CellShapeTagEmpty. This type of cell has no points, edges, or faces and can be thought of as a placeholder for a null or void cell.

There is also a special cell shape "tag" named vtkm::CellShapeTagGeneric that is used when the actual cell shape is not known at compile time. CellShapeTagGeneric actually has a member variable named Id that stores the identifier for the cell shape. There is no equivalent identifier for a generic cell; cell shape identifiers can be placed in a vtkm::IdComponent at runtime.

When using cell shapes in templated classes and functions, you can use the VTKM_IS_CELL_SHAPE_TAG to ensure a type is a valid cell shape tag. This macro takes one argument and will produce a compile error if the argument is not a cell shape tag type.

Converting Between Tags and Identifiers

Every cell shape tag has a member variable named Id that contains the identifier for the cell shape. This provides a convenient mechanism for converting a cell shape tag to an identifier. Most cell shape tags have their `Id` member as a compile-time constant, but `CellShapeTagGeneric` is set at run time.

vtkm/CellShape.h also declares a templated class named `vtkm::CellShapeIdToTag` that converts a cell shape identifier to a cell shape tag. `CellShapeIdToTag` as a single template argument that is the identifier. Inside the class is a type named `Tag` that is the type of the correct tag.

Example 8.4: Using `CellShapeIdToTag`.

```
void CellFunction(vtkm::CellShapeTagTriangle)
{
  std::cout << "In CellFunction for triangles." << std::endl;
}

void DoSomethingWithACell()
{
  // Calls CellFunction overloaded with a vtkm::CellShapeTagTriangle.
  CellFunction(vtkm::CellShapeIdToTag<vtkm::CELL_SHAPE_TRIANGLE>::Tag());
}
```

However, `CellShapeIdToTag` is only viable if the identifier can be resolved at compile time. In the case where a cell identifier is stored in a variable or an array or the code is using a `CellShapeTagGeneric`, the correct cell shape is not known at run time. In this case, `vtkmGenericCellShapeMacro` can be used to check all possible conditions. This macro is embedded in a switch statement where the condition is the cell shape identifier. `vtk-mGenericCellShapeMacro` has a single argument, which is an expression to be executed. Before the expression is executed, a type named `CellShapeTag` is defined as the type of the appropriate cell shape tag. Often this method is used to implement the condition for a `CellShapeTagGeneric` in a function overloaded for cell types. A demonstration of `vtkm-GenericCellShapeMacro` is given in Example 8.5.

Cell Traits

The vtkm/CellTraits.h header file contains a traits class named `vtkm::CellTraits` that provides information about a cell. Each specialization of `CellTraits` contains the following members.

`TOPOLOGICAL_DIMENSIONS` Defines the topological dimensions of the cell type. This is 3 for polyhedra, 2 for polygons, 1 for lines, and 0 for points.

`TopologicalDimensionsTag` A type set to either `vtkm::CellTopologicalDimensionsTag`<3>, `CellTopologicalDimensionsTag`<2>, `CellTopologicalDimensionsTag`<1>, or `CellTopologicalDimensionsTag`<0>. The number is consistent with `TOPOLOGICAL_DIMENSIONS`. This tag is provided for convenience when specializing functions.

`IsSizeFixed` Set to either `vtkm::CellTraitsTagSizeFixed` for cell types with a fixed number of points (for example, triangle) or `vtkm::CellTraitsTagSizeVariable` for cell types with a variable number of points (for example, polygon).

NUM_POINTS A vtkm::IdComponent set to the number of points in the cell. This member is only defined when there is a constant number of points (i.e. IsSizeFixed is set to vtkm::CellTraitsTagSizeFixed).

Example 8.5: Using CellTraits to implement a polygon normal estimator.

```
namespace detail {

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector, typename WorkletType>
VTKM_EXEC_CONT_EXPORT
typename PointCoordinatesVector::ComponentType
CellNormalImpl(const PointCoordinatesVector &pointCoordinates,
               vtkm::CellTopologicalDimensionsTag<2>,
               const WorkletType &worklet)
{
  if (pointCoordinates.GetNumberOfComponents() >= 3)
  {
    return vtkm::TriangleNormal(pointCoordinates[0],
                                pointCoordinates[1],
                                pointCoordinates[2]);
  }
  else
  {
    worklet.RaiseError("Degenerate polygon.");
    return typename PointCoordinatesVector::ComponentType();
  }
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector,
         vtkm::IdComponent Dimensions,
         typename WorkletType>
VTKM_EXEC_CONT_EXPORT
typename PointCoordinatesVector::ComponentType
CellNormalImpl(const PointCoordinatesVector &,
               vtkm::CellTopologicalDimensionsTag<Dimensions>,
               const WorkletType &worklet)
{
  worklet.RaiseError("Only polygons supported for cell normals.");
  return typename PointCoordinatesVector::ComponentType();
}

} // namespace detail

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename CellShape,
         typename PointCoordinatesVector,
         typename WorkletType>
VTKM_EXEC_CONT_EXPORT
typename PointCoordinatesVector::ComponentType
CellNormal(CellShape,
           const PointCoordinatesVector &pointCoordinates,
           const WorkletType &worklet)
{
  return detail::CellNormalImpl(
        pointCoordinates,
        typename vtkm::CellTraits<CellShape>::TopologicalDimensionsTag(),
        worklet);
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector,
         typename WorkletType>
VTKM_EXEC_CONT_EXPORT
typename PointCoordinatesVector::ComponentType
CellNormal(vtkm::CellShapeTagGeneric shape,
           const PointCoordinatesVector &pointCoordinates,
           const WorkletType &worklet)
{
  switch(shape.Id)
  {
    vtkmGenericCellShapeMacro(
          return CellNormal(CellShapeTag(), pointCoordinates, worklet));
    default:
      worklet.RaiseError("Unknown cell type.");
      return typename PointCoordinatesVector::ComponentType();
  }
}
```

8.3.2 Parametric and World Coordinates

Each cell type supports a one-to-one mapping between a set of parametric coordinates in the unit cube (or some subset of it) and the points in 3D space that are the locus contained in the cell. Parametric coordinates are useful because certain features of the cell, such as vertex location and center, are at a consistent location in parametric space irrespective of the location and distortion of the cell in world space. Also, many field operations are much easier with parametric coordinates.

The vtkm/exec/ParametricCoordinates.h header file contains the following functions for working with parametric coordinates.

vtkm::exec::ParametricCoordinatesCenter Returns the parametric coordinates for the center of a given shape. It takes 4 arguments: the number of points in the cell, a `vtkm::Vec` of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a `vtkm::Vec<vtkm::FloatDefault,3>` instead of passing it as a parameter.

vtkm::exec::ParametricCoordinatesPoint Returns the parametric coordinates for a given point of a given shape. It takes 5 arguments: the number of points in the cell, the index of the point to query, a `vtkm::Vec` of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a `vtkm::Vec<vtkm::FloatDefault,3>` instead of passing it as a parameter.

vtkm::exec::ParametricCoordinatesToWorldCoordinates Given a vector of point coordinates (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors), returns the world coordinates.

vtkm::exec::WorldCoordinatesToParametricCoordinates Given a vector of point coordinates (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size

3 containing world coordinates, a shape tag, and a worklet object (for raising errors), returns the parametric coordinates. This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

8.3.3 Interpolation

The shape of every cell is defined by the connections of some finite set of points. Field values defined on those points can be interpolated to any point within the cell to estimate a continuous field.

The vtkm/exec/CellInterpolate.h header contains the function `vtkm::exec::CellInterpolate` that takes a vector of point field values (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors). It returns the field interpolated to the location represented by the given parametric coordinates.

Example 8.6: Interpolating field values to a cell's center.

```
struct CellCenters : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(TopologyIn,
                                FieldInPoint<> inputField,
                                FieldOutCell<> outputField);
  typedef void ExecutionSignature(CellShape, PointCount, _2, _3);
  typedef _1 InputDomain;

  template<typename CellShapeTag,typename FieldInVecType,typename FieldOutType>
  VTKM_EXEC_EXPORT
  void operator()(CellShapeTag shape,
                  vtkm::IdComponent pointCount,
                  const FieldInVecType &inputField,
                  FieldOutType &outputField) const
  {
    vtkm::Vec<vtkm::FloatDefault,3> center =
        vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField = vtkm::exec::CellInterpolate(inputField, center, shape, *this);
  }
};
```

8.3.4 Derivatives

Since interpolations provide a continuous field function over a cell, it is reasonable to consider the derivative of this function. The vtkm/exec/CellDerivative.h header contains the function `vtkm::exec::CellDerivative` that takes a vector of scalar point field values (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors). It returns the field derivative at the location represented by the given parametric coordinates. The derivative is return in a `vtkm::Vec` of size 3 corresponding to the partial derivatives in the x, y, and z directions. This derivative is equivalent to the gradient of the field.

Example 8.7: Computing the derivative of the field at cell centers.

```
struct CellDerivatives : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(TopologyIn,
                                FieldInPoint<> inputField,
                                FieldInPoint<Vec3> pointCoordinates,
                                FieldOutCell<> outputField);
  typedef void ExecutionSignature(CellShape, PointCount, _2, _3, _4);
  typedef _1 InputDomain;

  template<typename CellShapeTag,
           typename FieldInVecType,
           typename PointCoordVecType,
           typename FieldOutType>
  VTKM_EXEC_EXPORT
  void operator()(CellShapeTag shape,
                  vtkm::IdComponent pointCount,
                  const FieldInVecType &inputField,
                  const PointCoordVecType &pointCoordinates,
                  FieldOutType &outputField) const
  {
    vtkm::Vec<vtkm::FloatDefault,3> center =
        vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField = vtkm::exec::CellDerivative(inputField,
                                             pointCoordinates,
                                             center,
                                             shape,
                                             *this);
  }
};
```

Chapter 9

Advanced Worklet Customization

Chapter 7 describes the basics of creating and using worklets. Many visualization algorithms can be implemented using VTK-m's existing worklet types and features. However, new algorithms and designs may require features not provided by VTK-m's current worklet set. In such cases it is possible to directly design filters using the lower level device adapter operations [AS DESCRIBED IN SECTION BLA]. But by adding features to the worklet mechanisms, new designs can be integrated better with the other VTK-m features and can be repurposed in interesting ways for other algorithms.

This chapter provides the information necessary to create new mechanisms for worklets. It first describes the interface for getting data from the control environment objects to the data passed to a worklet invocation and back. It then describes how to modify these mechanisms to create new data movement structures and new worklet types.

9.1 Transferring Arguments from Control to Execution

From the `ControlSignature` and `ExecutionSignature` defined in worklets, VTK-m uses template meta-programming to build the code required to manage data from control to execution environment. This management is handled by three classes that provide type checking, transportation, and fetching.

[I'VE BEEN THINKING THAT ONE MORE FEATURE THAT THESE CLASSES SHOULD PROVIDE IS THE ABILITY TO RETURN THE SIZE OF THE DOMAIN. THAT WOULD MAKE THINGS SIMPLER AND SAFER FOR GETTING THE INPUT DOMAIN SIZE AND CHECKING THE REMAINING DOMAIN SIZES.]

9.1.1 Type Checks

Before attempting to move data from the control to the execution environment, the VTK-m dispatchers check the input types to ensure that they are compatible with the associated `ControlSignature` concept. This is done with the `vtkm::cont::arg::TypeCheck` struct.

The TypeCheck struct is templated with two parameters. The first parameter is a tag that identifies which check to perform. The second parameter is the type of the control argument (after any dynamic casts). The TypeCheck class contains a static constant Boolean named value that is true if the type in the second parameter is compatible with the tag in the first or false otherwise.

Type checks are implemented with a defined type check tag (which, by convention, is defined in the vtkm::cont::arg namespace and starts with TypeCheckTag) and a partial specialization of the vtkm::cont::arg::TypeCheck structure. The following type checks (identified by their tags) are provided in VTK-m.

vtkm::cont::arg::TypeCheckTagArray True if the type is a vtkm::cont::ArrayHandle. TypeCheckTagArray also has a template parameter that is a type list. The ArrayHandle must also have a value type contained in this type list.

vtkm::cont::arg::TypeCheckTagExecObject True if the type is an execution object. All execution objects must derive from vtkm::exec::ExecutionObjectBase and must be copyable through memcpy or similar mechanism.

Here are some trivial examples of using TypeCheck. Typically these checks are done internally in the base VTK-m dispatcher code, so these examples are for demonstration only.

Example 9.1: Behavior of vtkm::cont::arg::TypeCheck.

```
struct MyExecObject : vtkm::exec::ExecutionObjectBase { vtkm::Id Value; };

void DoTypeChecks()
{
  using vtkm::cont::arg::TypeCheck;
  using vtkm::cont::arg::TypeCheckTagArray;
  using vtkm::cont::arg::TypeCheckTagExecObject;

  bool check1 = TypeCheck<TypeCheckTagExecObject, MyExecObject>::value; // true
  bool check2 = TypeCheck<TypeCheckTagExecObject, vtkm::Id>::value;     // false

  typedef vtkm::cont::ArrayHandle<vtkm::Float32> ArrayType;

  bool check3 =                                                         // true
      TypeCheck<TypeCheckTagArray<vtkm::TypeListTagField>, ArrayType>::value;
  bool check4 =                                                         // false
      TypeCheck<TypeCheckTagArray<vtkm::TypeListTagIndex>, ArrayType>::value;
  bool check5 = TypeCheck<TypeCheckTagExecObject, ArrayType>::value;    // false
}
```

9.1.2 Transport

After all the argument types are checked, the base dispatcher must load the data into the execution environment before scheduling a job to run there. This is done with the vtkm::cont::arg::Transport struct.

The `Transport` `struct` is templated with three parameters. The first parameter is a tag that identifies which transport to perform. The second parameter is the type of the control parameter (after any dynamic casts). The third parameter is a device adapter tag for the device on which the data will be loaded.

A `Transport` contains a `typedef` named `ExecObjectType` that is the type used after data is moved to the execution environment. A `Transport` also has a `const` parenthesis operator that takes the control-side object and the size of the domain and returns an execution-side object. This operator is called in the control environment, and the returned object must be ready to be passed to the execution environment.

Transports are implemented with a defined transport tag (which, by convention, is defined in the `vtkm::cont::arg` namespace and starts with `TransportTag`) and a partial specialization of the `vtkm::cont::arg::Transport` structure. The following transports (identified by their tags) are provided in VTK-m.

`vtkm::cont::arg::TransportTagArrayIn` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInput` method. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagArrayOut` Allocates data onto the specified device for a `vtkm::cont::ArrayHandle` using the array handle's `PrepareForOutput` method. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagExecObject` Simply returns the given execution object, which should be ready to load onto the device.

Here are some trivial examples of using `Transport`. Typically this movement is done internally in the base VTK-m dispatcher code, so these examples are for demonstration only.

Example 9.2: Behavior of `vtkm::cont::arg::Transport`.

```
struct MyExecObject : vtkm::exec::ExecutionObjectBase { vtkm::Id Value; };

typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType;

void DoTransport(const MyExecObject &inExecObject,
                 const ArrayType &inArray,
                 const ArrayType &outArray)
{
  typedef VTKM_DEFAULT_DEVICE_ADAPTER_TAG Device;

  using vtkm::cont::arg::Transport;
  using vtkm::cont::arg::TransportTagArrayIn;
  using vtkm::cont::arg::TransportTagArrayOut;
  using vtkm::cont::arg::TransportTagExecObject;

  // The executive object transport just passes the object through.
  typedef Transport<TransportTagExecObject,MyExecObject,Device>
      ExecObjectTransport;
  MyExecObject passedExecObject = ExecObjectTransport()(inExecObject, 10);
  // The array in transport returns a read-only array portal.
  typedef Transport<TransportTagArrayIn,ArrayType,Device> ArrayInTransport;
  ArrayInTransport::ExecObjectType inPortal = ArrayInTransport()(inArray, 10);

  // The array out transport returns an allocated array portal.
  typedef Transport<TransportTagArrayOut,ArrayType,Device> ArrayOutTransport;
  ArrayOutTransport::ExecObjectType outPortal =ArrayOutTransport()(outArray,10);
}
```

9.1.3 Fetch

Before the function of a worklet is invoked, the VTK-m internals pull the appropriate data out of the execution object and pass it to the worklet function. A class named vtkm::-exec::arg::Fetch is responsible for pulling this data out and putting computed data in to the execution objects.

The Fetch struct is templated with four parameters. The first parameter is a tag that identifies which type of fetch to perform. The second parameter is a different tag that identifies the aspect of the data to fetch. The third parameter is an Invocation type that provides details about how the worklet is being dispatched including a list of execution object parameters passed to the invocation. The fourth parameter is a vtkm::IdComponent that points to the invocation parameter that the data should be fetched from.

A Fetch contains a typedef named ValueType that is the type of data that is passed to and from the worklet function. A Fetch also has a pair of methods named Load and Store that get data from and add data to the execution object at a given domain or thread index.

Fetches are specified with a pair of fetch and aspect tags. Fetch tags are by convention defined in the vtkm::exec::arg namespace and start with FetchTag. Likewise, aspect tags are also defined in the vtkm::exec::arg namespace and start with AspectTag. The Fetch typedef is partially specialized on these two tags.

The most common aspect tag is vtkm::exec::arg::AspectTagDefault, and all fetch tags should have a specialization of vtkm::exec::arg::Fetch with this tag. The following list of fetch tags describes the execution objects they work with and the data they pull for each aspect tag they support.

[Don't forget to add index entries for both fetch and aspect where appropriate.]

vtkm::exec::arg::FetchTagArrayDirectIn Loads data from an array portal. This fetch only supports the AspectTagDefault aspect. The Load gets data directly from the domain (thread) index. The Store does nothing.

vtkm::exec::arg::FetchTagArrayDirectOut Stores data to an array portal. This fetch only supports the AspectTagDefault aspect. The Store sets data directly to the domain (thread) index. The Load does nothing.

vtkm::exec::arg::FetchTagExecObject Simply returns an execution object. This fetch only supports the `AspectTagDefault` aspect. The `Load` returns the executive object in the associated parameter. The `Store` does nothing.

In addition to the aforementioned aspect tags that are explicitly paired with fetch tags, VTK-m also provides some aspect tags that either modify the behavior of a general fetch or simply ignore the type of fetch.

vtkm::exec::arg::AspectTagWorkIndex Simply returns the domain (or thread) index ignoring any associated data. This aspect is used to implement the `WorkIndex` execution signature tag.

9.2 Function Interface Objects

For flexibility's sake a worklet is free to declare a `ControlSignature` with whatever number of arguments are sensible for its operation. The `Invoke` method of the dispatcher is expected to support arguments that match these arguments, and part of the dispatching operation may require these arguments to be augmented before the worklet is scheduled. This leaves dispatchers with the tricky task of managing some collection of arguments of unknown size and unknown types.

[FUNCTIONINTERFACE IS IN THE VTKM::INTERNAL INTERFACE. I STILL CAN'T DECIDE IF IT SHOULD BE MOVED TO THE VTKM INTERFACE.]

To simplify this management, VTK-m has the `vtkm::internal::FunctionInterface` class. `FunctionInterface` is a templated class that manages a generic set of arguments and return value from a function. An instance of `FunctionInterface` holds an instance of each argument. You can apply the arguments in a `FunctionInterface` object to a functor of a compatible prototype, and the resulting value of the function call is saved in the `FunctionInterface`.

9.2.1 Declaring and Creating

`vtkm::internal::FunctionInterface` is a templated class with a single parameter. The parameter is the *signature* of the function. A signature is a function type. The syntax in C++ is the return type followed by the argument types encased in parentheses.

Example 9.3: Declaring `vtkm::internal::FunctionInterface`.

```
// FunctionInterfaces matching some common POSIX functions.
vtkm::internal::FunctionInterface<size_t(const char *)>
    strlenInterface;
```

```
  vtkm::internal::FunctionInterface<char *(char *, const char *s2, size_t)>
      strncpyInterface;
```

The vtkm::internal::make_FunctionInterface function provies an easy way to create a FunctionInterface and initialize the state of all the parameters. make_FunctionInterface takes a variable number of arguments, one for each parameter. Since the return type is not specified as an argument, you must always specify it as a template parameter.

Example 9.4: Using vtkm::internal::make_FunctionInterface.

```
  const char *s = "Hello World";
  static const size_t BUFFER_SIZE = 100;
  char *buffer = (char *)malloc(BUFFER_SIZE);

  strlenInterface =
      vtkm::internal::make_FunctionInterface<size_t>(s);

  strncpyInterface =
      vtkm::internal::make_FunctionInterface<char *>(buffer, s, BUFFER_SIZE);
```

9.2.2 Parameters

One created, FunctionInterface contains methods to query and manage the parameters and objects associated with them. The number of parameters can be retrieved either with the constant field ARITY or with the GetArity method.

Example 9.5: Getting the arity of a FunctionInterface.

```
  BOOST_STATIC_ASSERT(
        vtkm::internal::FunctionInterface<size_t(const char *)>::ARITY == 1);

  vtkm::IdComponent arity = strncpyInterface.GetArity();  // arity = 3
```

To get a particular parameter, FunctionInterface has the templated method GetParameter. The template parameter is the index of the parameter. Note that the parameters in FunctionInterface start at index 1. Although this is uncommon in C++, it is customary to number function arguments starting at 1.

There are two ways to specify the index for GetParameter. The first is to directly specify the template parameter (e.g. GetParameter<1>()). However, note that in a templated function or method where the type is not fully resolved the compiler will not register GetParameter as a templated method and will fail to parse the template argument without a template keyword. The second way to specify the index is to provide a vtkm::internal::IndexTag object as an argument to GetParameter. Although this syntax is more verbose, it works the same whether the FunctionInterface is fully resolved or not. The following example shows both methods in action.

Example 9.6: Using FunctionInterface::GetParameter().

```
void GetFirstParameterResolved(
    const vtkm::internal::FunctionInterface<void(std::string)> &interface)
```

```
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>())
            << std::endl;
}

template<typename FunctionSignature>
void GetFirstParameterTemplated(
    const vtkm::internal::FunctionInterface<FunctionSignature> &interface)
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.template GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>())
            << std::endl;
}
```

Likewise, there is a `SetParmeter` method for changing parameters. The same rules for indexing and template specification apply.

Example 9.7: Using `FunctionInterface::SetParameter()`.

```
void SetFirstParameterResolved(
    vtkm::internal::FunctionInterface<void(std::string)> &interface,
    const std::string &newFirstParameter)
{
  // The following two uses of SetParameter are equivalent
  interface.SetParameter<1>(newFirstParameter);
  interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
}

template<typename FunctionSignature, typename T>
void SetFirstParameterTemplated(
    vtkm::internal::FunctionInterface<FunctionSignature> &interface,
    T newFirstParameter)
{
  // The following two uses of SetParameter are equivalent
  interface.template SetParameter<1>(newFirstParameter);
  interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
}
```

9.2.3 Invoking

`FunctionInterface` can invoke a functor of a matching signature using the parameters stored within. If the functor returns a value, that return value will be stored in the `FunctionInterface` object for later retrieval. There are several versions of the invoke method. There are always seperate versions of invoke methods for the control and execution environments so that functors for either environment can be executed. The basic version of invoke passes the parameters directly to the function and directly stores the result.

Example 9.8: Invoking a `FunctionInterface`.

```
  vtkm::internal::FunctionInterface<size_t(const char *)> strlenInterface;
  strlenInterface.SetParameter<1>("Hello world");

  strlenInterface.InvokeCont(strlen);

  size_t length = strlenInterface.GetReturnValue();   // length = 11
```

Another form of the invoke methods takes a second transform functor that is applied to each argument before passed to the main function. If the main function returns a value, the transform is applied to that as well before being stored back in the FunctionInterface.

Example 9.9: Invoking a FunctionInterface with a transform.

```
// Our transform converts C strings to integers, leaves everything else alone.
struct TransformFunctor
{
  template<typename T>
  VTKM_CONT_EXPORT
  const T &operator()(const T &x) const
  {
    return x;
  }

  VTKM_CONT_EXPORT
  const vtkm::Int32 operator()(const char *x) const
  {
    return atoi(x);
  }
};

// The function we are invoking simply compares two numbers.
struct IsSameFunctor
{
  template<typename T1, typename T2>
  VTKM_CONT_EXPORT
  bool operator()(const T1 &x, const T2 &y) const
  {
    return x == y;
  }
};

void TryTransformedInvoke()
{
  vtkm::internal::FunctionInterface<bool(const char *, vtkm::Int32)>
      functionInterface =
        vtkm::internal::make_FunctionInterface<bool>((const char *)"42",
                                                     (vtkm::Int32)42);

  functionInterface.InvokeCont(IsSameFunctor(), TransformFunctor());

  bool isSame = functionInterface.GetReturnValue();     // isSame = true
}
```

As demonstrated in the previous examples, FunctionInterface has a method named GetReturnValue that returns the value from the last invoke. Care should be taken to only use GetReturnValue when the function specification has a return value. If the function signature has a void return type, using GetReturnValue will cause a compile error.

FunctionInterface has an alternate method named GetReturnValueSafe that returns the value wrapped in a templated structure named vtkm::internal::FunctionInterface-eReturnContainer. This structure always has a static constant Boolean named VALID that is false if there is no return type and true otherwise. If the container is valid, it also has an entry named Value containing the result.

Example 9.10: Getting return value from FunctionInterface safely.

```
template<typename ResultType, bool Valid> struct PrintReturnFunctor;

template<typename ResultType>
```

```
struct PrintReturnFunctor<ResultType, true>
{
  VTKM_CONT_EXPORT
  void operator()(
      const vtkm::internal::FunctionInterfaceReturnContainer<ResultType> &x)
  const
  {
    std::cout << x.Value << std::endl;
  }
};

template<typename ResultType>
struct PrintReturnFunctor<ResultType, false>
{
  VTKM_CONT_EXPORT
  void operator()(
      const vtkm::internal::FunctionInterfaceReturnContainer<ResultType> &)
  const
  {
    std::cout << "No return type." << std::endl;
  }
};

template<typename FunctionInterfaceType>
void PrintReturn(const FunctionInterfaceType &functionInterface)
{
  typedef typename FunctionInterfaceType::ResultType ResultType;
  typedef vtkm::internal::FunctionInterfaceReturnContainer<ResultType>
      ReturnContainerType;

  PrintReturnFunctor<ResultType, ReturnContainerType::VALID> printReturn;
  printReturn(functionInterface.GetReturnValueSafe());
}
```

9.2.4 Modifying Parameters

In addition to storing and querying parameters and invoking functions, `FunctionInterface` also contains multiple ways to modify the parameters to augment the function calls. This can be used in the same use case as a chain of function calls that generally pass their parameters but also augment the data along the way.

The `Append` method returns a new `FunctionInterface` object with the same parameters plus a new parameter (the argument to `Append`) to the end of the parameters. There is also a matching `AppendType` templated structure that can return the type of an augmented `FunctionInterface` with a new type appended.

Example 9.11: Appending parameters to a `FunctionInterface`.

```
  using vtkm::internal::FunctionInterface;
  using vtkm::internal::make_FunctionInterface;

  typedef FunctionInterface<void(std::string, vtkm::Id)>
      InitialFunctionInterfaceType;
  InitialFunctionInterfaceType initialFunctionInterface =
      make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

  typedef FunctionInterface<void(std::string, vtkm::Id, std::string)>
      AppendedFunctionInterfaceType1;
  AppendedFunctionInterfaceType1 appendedFunctionInterface1 =
      initialFunctionInterface.Append(std::string("foobar"));
  // appendedFunctionInterface1 has parameters ("Hello World", 42, "foobar")

  typedef InitialFunctionInterfaceType::AppendType<vtkm::Float32>::type
      AppendedFunctionInterfaceType2;
  AppendedFunctionInterfaceType2 appendedFunctionInterface2 =
      initialFunctionInterface.Append(vtkm::Float32(3.141));
  // appendedFunctionInterface2 has parameters ("Hello World", 42, 3.141)
```

Replace is a similar method that returns a new FunctionInterface object with the same paraemters except with a specified parameter replaced with a new parameter (the argument to Replace). There is also a matching ReplaceType templated structure that can return the type of an augmented FunctionInterface with one of the parameters replaced.

Example 9.12: Replacing parameters in a FunctionInterface.

```
  using vtkm::internal::FunctionInterface;
  using vtkm::internal::make_FunctionInterface;

  typedef FunctionInterface<void(std::string, vtkm::Id)>
      InitialFunctionInterfaceType;
  InitialFunctionInterfaceType initialFunctionInterface =
      make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

  typedef FunctionInterface<void(vtkm::Float32, vtkm::Id)>
      ReplacedFunctionInterfaceType1;
  ReplacedFunctionInterfaceType1 replacedFunctionInterface1 =
      initialFunctionInterface.Replace<1>(vtkm::Float32(3.141));
  // replacedFunctionInterface1 has parameters (3.141, 42)

  typedef InitialFunctionInterfaceType::ReplaceType<2, std::string>::type
      ReplacedFunctionInterfaceType2;
  ReplacedFunctionInterfaceType2 replacedFunctionInterface2 =
      initialFunctionInterface.Replace<2>(std::string("foobar"));
  // replacedFunctionInterface2 has parameters ("Hello World", "foobar")
```

It is sometimes desirable to make multiple modifications at a time. This can be achieved by chaining modifications by calling Append or Replace on the result of a previous call.

Example 9.13: Chaining Replace and Append with a FunctionInterface.

```
template<typename FunctionInterfaceType>
void FunctionCallChain(const FunctionInterfaceType &parameters,
                       vtkm::Id arraySize)
{
  // In this hypothetical function call chain, this function replaces the
  // first parameter with an array of that type and appends the array size
  // to the end of the parameters.

  typedef typename FunctionInterfaceType::template ParameterType<1>::type
      ArrayValueType;

  // Allocate and initialize array.
  ArrayValueType value = parameters.template GetParameter<1>();
  ArrayValueType *array = new ArrayValueType[arraySize];
  for (vtkm::Id index = 0; index < arraySize; index++)
  {
    array[index] = value;
  }

  // Call next function with modified parameters.
  NextFunctionChainCall(
        parameters.template Replace<1>(array).Append(arraySize));
  // Clean up.
  delete[] array;
}
```

9.2.5 Transformations

Rather than replace a single item in a FunctionInterface, it is sometimes desirable to change them all in a similar way. FunctionInterface supports two basic transform operations on its parameters: a static transform and a dynamic transform. The static transform determines its types at compile-time whereas the dynamic transform happens at run-time.

The static transform methods (named StaticTransformCont and StaticTransformExec) operate by accepting a functor that defines a function with two arguments. The first argument is the FunctionInterface parameter to transform. The second argument is an instance of the vtkm::internal::IndexTag templated class that statically identifies the parameter index being transformed. An IndexTag object has no state, but the class contains a static integer named INDEX. The function returns the transformed argument.

The functor must also contain a templated class named ReturnType with an internal type named type that defines the return type of the transform for a given parameter type. ReturnType must have two template parameters. The first template parameter is the type of the FunctionInterface parameter to transform. It is the same type as passed to the operator. The second template parameter is a vtkm::IdComponent specifying the index.

The transformation is only applied to the parameters of the function. The return argument is unaffected.

The return type can be determined with the StaticTransformType template in the FunctionInterface class. StaticTransformType has a single parameter that is the transform functor and contains a type named type that is the transformed FunctionInterface.

In the following example, a static transform is used to convert a FunctionInterface to a new object that has the pointers to the parameters rather than the values themselves. The parameter index is always ignored as all parameters are uniformly transformed.

Example 9.14: Using a static transform of function interface class.

```
struct ParametersToPointersFunctor {
  template<typename T, vtkm::IdComponent Index>
  struct ReturnType {
    typedef const T *type;
  };

  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  const T *operator()(const T &x, vtkm::internal::IndexTag<Index>) const {
    return &x;
  }
};

template<typename FunctionInterfaceType>
```

```
VTKM_CONT_EXPORT
typename FunctionInterfaceType::
    template StaticTransformType<ParametersToPointersFunctor>::type
ParametersToPointers(const FunctionInterfaceType &functionInterface)
{
  return functionInterface.StaticTransformCont(ParametersToPointersFunctor());
}
```

There are cases where one set of parameters must be transformed to another set, but the types of the new set are not known until run-time. That is, the transformed type depends on the contents of the data. The `DynamicTransformCont` method achieves this using a templated callback that gets called with the correct type at run-time.

The dynamic transform works with two functors provided by the user code (as opposed to the one functor in static transform). These functors are called the transform functor and the finish functor. The transform functor accepts three arguments. The first argument is a parameter to transform. The second argument is a continue function. Rather than return the transformed value, the transform functor calls the continue function, passing the transformed value as an argument. The third argument is a `vtkm::internal::IndexTag` for the index of the argument being transformed.

Unlike its static counterpart, the dynamic transform method does not return the transformed `FunctionInterface`. Instead, it passes the transformed `FunctionInterface` to the finish functor passed into `DynamicTransformCont`.

In the following contrived but illustrative example, a dynamic transform is used to convert strings containing numbers into number arguments. Strings that do not have numbers and all other arguments are passed through. Note that because the types for strings are not determined till run-time, this transform cannot be determined at compile time with meta-template programming. The index argument is ignored because all arguments are transformed the same way.

Example 9.15: Using a dynamic transform of a function interface.

```
struct UnpackNumbersTransformFunctor {
  template<typename InputType,
           typename ContinueFunctor,
           vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  void operator()(const InputType &input,
                  const ContinueFunctor &continueFunction,
                  vtkm::internal::IndexTag<Index>) const
  {
    continueFunction(input);
  }

  template<typename ContinueFunctor, vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  void operator()(const std::string &input,
                  const ContinueFunctor &continueFunction,
                  vtkm::internal::IndexTag<Index>) const
  {
    if ((input[0] >= '0') && (input[0] <= '9'))
    {
      std::stringstream stream(input);
      vtkm::FloatDefault value;
      stream >> value;
      continueFunction(value);
    }
    else
    {
      continueFunction(input);
    }
  }
};

struct UnpackNumbersFinishFunctor {
  template<typename FunctionInterfaceType>
  VTKM_CONT_EXPORT
  void operator()(FunctionInterfaceType &functionInterface) const
  {
    // Do something
  }
};

template<typename FunctionInterfaceType>
void DoUnpackNumbers(const FunctionInterfaceType &functionInterface)
{
  functionInterface.DynamicTransformCont(UnpackNumbersTransformFunctor(),
                                         UnpackNumbersFinishFunctor());
}
```

One common use for the FunctionInterface dynamic transform is to convert parameters of virtual polymorphic type like vtkm::cont::DynamicArrayHandle and vtkm::cont::-DynamicPointCoordinates. This use case is handled with a functor named vtkm::cont::-internal::DynamicTransform. When used as the dynamic transform functor, it will convert all of these dynamic types to their static counterparts.

Example 9.16: Using DynamicTransform to cast dynamic arrays in a function interface.

```
template<typename Device>
struct ArrayCopyFunctor {
  template<typename Signature>
  VTKM_CONT_EXPORT
  void operator()(
      vtkm::internal::FunctionInterface<Signature> functionInterface) const
  {
    functionInterface.InvokeCont(*this);
  }

  template<typename T, class CIn, class COut>
  VTKM_CONT_EXPORT
  void operator()(const vtkm::cont::ArrayHandle<T, CIn> &input,
                  vtkm::cont::ArrayHandle<T, COut> &output) const
  {
    vtkm::cont::DeviceAdapterAlgorithm<Device>::Copy(input, output);
  }

  template<typename TIn, typename TOut, class CIn, class COut>
  VTKM_CONT_EXPORT
  void operator()(const vtkm::cont::ArrayHandle<TIn, CIn> &,
                  vtkm::cont::ArrayHandle<TOut, COut> &) const
  {
    throw vtkm::cont::ErrorControlBadType(
          "Arrays to copy must be the same type.");
  }
};

template<typename Device>
void CopyDynamicArrays(vtkm::cont::DynamicArrayHandle input,
                       vtkm::cont::DynamicArrayHandle output,
                       Device)
```

```
{
  vtkm::internal::FunctionInterface<void(vtkm::cont::DynamicArrayHandle,
                                         vtkm::cont::DynamicArrayHandle)>
      functionInterface =
        vtkm::internal::make_FunctionInterface<void>(input, output);

  functionInterface.DynamicTransformCont(
        vtkm::cont::internal::DynamicTransform(), ArrayCopyFunctor<Device>());
}
```

9.2.6 For Each

The invoke methods (principally) make a single function call passing all of the parameters to this function. The transform methods call a function on each parameter to convert it to some other data type. It is also sometimes helpful to be able to call a unary function on each parameter that is not expected to return a value. Typically the use case is for the function to have some sort of side effect. For example, the function might print out some value (such as in the following example) or perform some check on the data and throw an exception on failure.

This feature is implemented in the for each methods of `FunctionInterface`. As with all the `FunctionInterface` methods that take functors, there are separate implementations for the control environment and the execution environment. There are also separate implementations taking `const` and non-`const` references to functors to simplify making functors with side effects.

Example 9.17: Using the `ForEach` feature of `FunctionInterface`.

```
struct PrintArgumentFunctor{
  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  void operator()(const T &argument, vtkm::internal::IndexTag<Index>) const
  {
    std::cout << Index << ":" << argument << " ";
  }
};

template<typename FunctionInterfaceType>
VTKM_CONT_EXPORT
void PrintArguments(const FunctionInterfaceType &functionInterface)
{
  std::cout << "( ";
  functionInterface.ForEachCont(PrintArgumentFunctor());
  std::cout << ")" << std::endl;
}
```

9.3 Invocation Objects

9.4 Creating New `ControlSignature` Tags

9.5 Creating New `ExecutionSignature` Tags

9.6 Creating New Worklet Types

9.6.1 New Worklet Superclasses

9.6.2 Dispatch Workflow

9.6.3 New Dispatch Classes

Chapter 10

OpenGL Interoperability

Chapter 11

Coding Conventions

Several developers contribute to VTK-m and we welcome others who are interested to also contribute to the project. To ensure readability and consistency in the code, we have adopted the following coding conventions. Many of these conventions are adapted from the coding conventions of the VTK project. This is because many of the developers are familiar with VTK coding and because we expect VTK-m to have continual interaction with VTK.

- All code contributed to VTK-m must be compatible with VTK-m's BSD license.
- Copyright notices should appear at the top of all source, configuration, and text files. The statement should have the following form:

```
//=============================================================================
//  Copyright (c) Kitware, Inc.
//  All rights reserved.
//  See LICENSE.txt for details.
//  This software is distributed WITHOUT ANY WARRANTY; without even
//  the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
//  PURPOSE.  See the above copyright notice for more information.
//
//  Copyright 2014 Sandia Corporation.
//  Copyright 2014 UT-Battelle, LLC.
//  Copyright 2014. Los Alamos National Security
//
//  Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation,
//  the U.S. Government retains certain rights in this software.
//
//  Under the terms of Contract DE-AC52-06NA25396 with Los Alamos National
//  Laboratory (LANL), the U.S. Government retains certain rights in
//  this software.
//=============================================================================
```

The CopyrightStatement test checks all files for a similar statement. The test will print out a suggested text that can be copied and pasted to any file that has a missing copyright statement (with appropriate replacement of comment prefix). Exceptions to this copyright statement (for example, third-party files with different but compatible statements) can be added to LICENSE.txt.

- All include files should use include guards. starting right after the copyright statement. The naming convention of the include guard macro is that it should start with `vtk_m` be followed with the path name, starting from the top-level source code directory under vtkm, with non alphanumeric characters, such as / and . replaced with underscores. The `#endif` part of the guard at the bottom of the file should include the guard name in a comment. For example, the vtkm/cont/ArrayHandle.h header contains the guard

  ```
  #ifndef vtk_m_cont_ArrayHandle_h
  #define vtk_m_cont_ArrayHandle_h
  ```

  at the top and

  ```
  #endif //vtk_m_cont_ArrayHandle_h
  ```

- VTK-m has several nested namespaces. The declaration of each namespace should be on its own line, and the code inside the namespace bracket should not be indented. The closing brace at the bottom of the namespace should be documented with a comment identifying the namespace. Namespaces can be grouped as desired. The following is a valid use of namespaces.

  ```
  namespace vtkm {
  namespace cont {

  namespace detail {

  class InternalClass;

  } // namespace detail

  class ExposedClass;

  }
  } // namespace vtkm::cont
  ```

- Multiple inheritance is not allowed in VTK-m classes.

- Any functional public class should be in its own header file with the same name as the class. The file should be in a directory that corresponds to the namespace the class is in. There are several exceptions to this rule.

  - Templated classes and template specialization often require the implementation of the class to be broken into pieces. Sometimes a specialization is placed in a header with a different name.
  - Many VTK-m toolkit features are not encapsulated in classes. Functions may be collected by purpose or co-located with associated class.
  - Although tags are technically classes, they behave as an enumeration for the compiler. Multiple tags that make up this enumeration are collected together.

  - Some classes, such as `vtkm::Vec` are meant to behave as basic types. These are sometimes collected together as if they were related `typedef`s. The `vtkm/Types.h` header is a good example of this.

- The indentation follows the Allman style. The curly brace (scope delimiter) for a block is placed on the line following the prototype or control statement and is indented with the outer scope (i.e. the curly brace does not line up with the code in the block). This differs from VTK style, but was agreed on by the developers as the more common style. Indentations are two spaces.

- Conditional clauses (including loop conditionals such as `for` and `while`) must be in braces below the conditional. That is, instead of

  ```
  if (test) { clause; }
  ```

  use

  ```
  if (test)
  {
    clause;
  }
  ```

  The rational for this requirement is to make it obvious whether the clause is executed when stepping through the code with the debugger. The one exception to this rule is when the clause contains a control-flow statement with obvious side effects such as `return` or `break`. However, even if the clause contains a single statement and is on the same line, the clause should be surrounded by braces.

- Use two space indentation.

- Tabs are not allowed. Only use spaces for indentation. No one can agree on what the size of a tab stop is, so it is better to not use them at all.

- There should be no trailing whitespace in any line.

- Use only alphanumeric characters in names. Use capitalization to demarcate words within a name (camel case). The exception is preprocessor macros and constant numbers that are, by convention, represented in all caps and a single underscore to demarcate words.

- Namespace names are in all lowercase. They should be a single word that designates its meaning.

- All class, method, member variable, and functions should start with a capital letter. Local variables should start in lower case and then use camel case. Exceptions can be made when such naming would conflict with previously established conventions in other library. (For example, `make_ArrayHandle` corresponds to `make_pair` in the standard template library.)

- Always spell out words in names; do not use abbreviations except in cases where the shortened form is widely understood and a name in its own right (e.g. OpenMP).

- Always use descriptive names in all identifiers, including local variable names. Particularly avoid meaningless names of a few characters (e.g. `x`, `foo`, or `tmp`) or numbered names with no meaning to the number or order (e.g. `value1`, `value2`,...). Also avoid the meaningless for loop variable names `i`, `j`, `k`, etc. Instead, use a name that identifies what type of index is being referenced such as `pointIndex`, `vertexIndex`, `componentIndex`, etc.

- Classes are documented with Doxygen-style comments before classes, methods, and functions.

- Exposed classes should not have public instance variables outside of exceptional situations. Access is given by convention through methods with names starting with `Set` and `Get` or through overloaded operators.

- References to classes and functions should be fully qualified with the namespace. This makes it easier to establish classes and functions from different packages and to find source and documentation for the referenced class. As an exception, if one class references an internal or detail class clearly associated with it, the reference can be shortened to `internal::` or `detail::`.

- use `this->` inside of methods when accessing class methods and instance variables to distinguish between local variables and instance variables.

- Include statements should generally be in alphabetical order. They can be grouped by package and type.

- Namespaces should not be brought into global scope or the scope of any VTK-m package namespace with the "using" keyword. It should also be avoided in class, method, and function scopes (fully qualified namespace references are preferred).

- All code must be valid by the C++03 and C++11 specifications. It must also compile on older compilers that support C++98. Code that uses language features not available in C++98 must have a second implementation that works around the limitations of C++98. The VTKM_FORCE_ANSI turns on a compiler check for ANSI compatibility in gcc and clang compilers.

- Limit all lines to 80 characters whenever possible.

- New code must include regression tests that will run on the dashboards. Generally a new class will have an associated "UnitTest" that will test the operation of the test directly. There may be other tests necessary that exercise the operation with different components or on different architectures.

- All code must compile and run without error or warning messages on the nightly dashboards, which should include Windows, Mac, and Linux.

- Use `vtkm::Id` in lieu of `int` or `long` for data structure indices and `vtkm::IdComponent` for component indices of `vtkm::Vec` and related classes (like `vtkm::exec::CellField` and `vtkm::math::Matrix`).

- Whenever possible, use templates to resolve data types like `float`, `double`, or vectors to make code as flexible as possible. If a specific data type is required, prefer the VTK-m–provided types like `vtkm::Float32` and `vtkm::Float64` over the standard C types like `float` or `double`. `vtkm::FloatDefault` can be used in cases where there is no reasonable way to specify data precision (for example, when generating coordinates for uniform grids), but should be use sparingly.

- All functions and methods defined within the Dax toolkit should be declared with `VTKM_CONT_EXPORT`, `VTKM_EXEC_EXPORT`, or `VTKM_EXEC_CONT_EXPORT`.

We should note that although these conventions impose a strict statute on VTK-m coding, these rules (other than those involving licensing and copyright) are not meant to be dogmatic. Examples can be found in the existing code that break these conventions, particularly when the conventions stand in the way of readability (which is the point in having them in the first place). For example, it is often the case that it is more readable for a complicated `typedef` to stretch a few characters past 80 even if it pushes past the end of a display.

Index

DISTRIBUTION:

1	MS 1326	Kenneth Moreland, 1461
1	MS 1327	Ron Oldfield, 01461
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories