SAND2015-4645R

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.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

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

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

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

1 Introduction **15**

2 Basic Provisions **17**

2.1 Package Structure 17

2.2 Function and Method Exports 18

2.3 Core Data Types 19

2.3.1 Single Number Types 19

2.3.2 Vector Types 20

2.3.3 Extents 22

2.3.4 Pair 23

2.4 Traits 24

2.4.1 Type Traits 24

2.4.2 Vector Traits 25

2.5 List Tags 27

2.5.1 Building List Tags 28

2.5.2 Type Lists 29

2.5.3 Operating on Lists 30

3 Provided Filters **33**

4 Control Environment **35**

4.1 Device Adapter Tag 35

4.1.1 Default Device Adapter 35

4.1.2 Specifying Device Adapter Tags 37
4.2 Array Handle 38
4.2.1 Creating Array Handles 38
4.2.2 Array Portals 40
4.2.3 Interface to Execution Environment 43
4.2.4 Basic Storage 44
4.2.5 Adapting Data Structures 45
4.2.6 Implicit Array Handles 49
4.2.7 Transformed Arrays 51
4.2.8 Permuted Arrays 53
4.2.9 Zipped Arrays 54
4.2.10 Derived Storage 55
4.3 Dynamic Array Handle 63
4.3.1 Querying and Casting 63
4.3.2 Casting to Unknown Types 64
4.3.3 Specifying Cast Lists 65
4.4 Dynamic Point Coordinates 66
4.4.1 Querying and Casting 67
4.4.2 Casting to Unknown Array Types 68
4.4.3 Specifying Cast Lists 68
4.4.4 Defining new Point Coordinate Types 69
4.5 Grid Structures 71
4.6 Timers 71
4.7 Error Handling 72
4.8 Device Adapter Algorithms 73

5 Execution Environment **77**

6 Worklets **79**

6.1 Worklet Types 79

6.2 Dispatchers 80

6.3 Provided Worklets 80

6.4 Creating Worklets 80

6.4.1 Control Signature 81

6.4.2 Execution Signature 83

6.4.3 Input Domain 83

6.4.4 Worklet Operator 84

6.4.5 Worklet Types 84

Field Map 84

7 Advanced Worklet Customization **87**

7.1 Transferring Arguments from Control to Execution 87

7.1.1 Type Checks 87

7.1.2 Transport 88

7.1.3 Fetch 90

7.2 Function Interface Objects 91

7.2.1 Declaring and Creating 91

7.2.2 Parameters 92

7.2.3 Invoking 93

7.2.4 Modifying Parameters 95

7.2.5 Transformations 97

7.2.6 For Each 100

7.3 Invocation Objects 101

7.4 Creating New `ControlSignature` Tags 101

7.5 Creating New `ExecutionSignature` Tags 101

7.6 Creating New Worklet Types 101

7.6.1 New Worklet Superclasses 101

7.6.2 Dispatch Workflow 101

7.6.3 New Dispatch Classes 101

8 OpenGL Interoperability **103**

9 Coding Conventions **105**

Index **110**

List of Figures

2.1 VTK-m package hierarchy. 18

List of Examples

2.1 Usage of export macro. 19

2.2 Creating vector types. 20

2.3 A Longer Vector. 21

2.4 Vector operations. 21

2.5 Repurposing a `vtkm::Vec`. 21

2.6 Creating and using an `Extent3`. 23

2.7 Definition of `vtkm::TypeTraits<vtkm::Float32>`. 24

2.8 Using `TypeTraits` for a generic remainder. 24

2.9 Definition of `vtkm::VecTraits<vtkm::Id3>`. 26

2.10 Using `VecTraits` for less functors. 26

2.11 Creating list tags. 28

2.12 Defining new type lists. 30

2.13 Converting dynamic types to static types with `ListForEach`. 30

4.1 Macros to port VTK-m code among different devices 36

4.2 Declaration of the `vtkm::cont::ArrayHandle` templated class. 38

4.3 Creating an `ArrayHandle` for output data. 39

4.4 Creating an `ArrayHandle` that points to a provided C array. 39

4.5 Creating an `ArrayHandle` that points to a provided `std::vector`. 39

4.6 Invalidating an `ArrayHandle` by letting the source `std::vector` leave scope. . 39

4.7 A simple array portal implementation. 40

4.8 Using `ArrayPortalToIterators`. 41

4.9 Using `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd`. 42

4.10 Using portals from an ArrayHandle. 42
4.11 Using an execution array portal from an ArrayHandle. 43
4.12 Declaration of the vtkm::cont::ArrayHandle templated class (again). 44
4.13 Specifying the storage type for an ArrayHandle. 45
4.14 Fictitious field storage used in custom array storage examples. 45
4.15 Array portal to adapt a third-party container to VTK-m. 46
4.16 Prototype for vtkm::cont::internal::Storage. 47
4.17 Storage to adapt a third-party container to VTK-m. 47
4.18 Array handle to adapt a third-party container to VTK-m. 48
4.19 Using an ArrayHandle with custom container. 49
4.20 Functor that doubles an index. 50
4.21 Declaring a ArrayHandleImplicit. 50
4.22 Using make_ArrayHandleImplicit. 50
4.23 Custom implicit array handle for even numbers. 50
4.24 Functor to scale and bias a value. 51
4.25 Using make_ArrayHandleTransform. 52
4.26 Custom transform array handle for scale and bias. 52
4.27 Using ArrayHandlePermutation. 53
4.28 Using make_ArrayHandlePermutation. 54
4.29 Using ArrayHandleZip. 54
4.30 Using make_ArrayHandleZip. 55
4.31 Derived array portal for concatenated arrays. 55
4.32 Storage for derived container of concatenated arrays. 56
4.33 Prototype for vtkm::cont::internal::ArrayTransfer. 59
4.34 Prototype for ArrayTransfer constructor. 59
4.35 ArrayTransfer for derived storage of concatenated arrays. 60

4.36 ArrayHandle for derived storage of concatenated arrays. 62
4.37 Creating a DynamicArrayHandle. 63
4.38 Querying the component and storage types of a DynamicArrayHandle. 63
4.39 Casting a DynamicArrayHandle to a concrete ArrayHandle. 64
4.40 Operating on DynamicArrayHandle with CastAndCall. 64
4.41 Trying all component types in a DynamicArrayHandle. 66
4.42 Specifying a single component type in a DynamicArrayHandle. 66
4.43 Specifying different storage types in a DynamicArrayHandle. 66
4.44 Specifying both component and storage types in a DynamicArrayHandle. 66
4.45 Creating DynamicPointCoordinates objects. 67
4.46 Querying the point coordinate type of DynamicPointCoordinates. 67
4.47 Casting DynamicPointCoordinates to concrete point coordinates. 68
4.48 Operating on DynamicPointCoordinates with CastAndCall. 68
4.49 A custom point coordinates class. 70
4.50 Using a custom point coordinates class. 70
4.51 Using vtkm::cont::Timer. 71
4.52 Simple error reporting. 72
4.53 Prototype for vtkm::cont::DeviceAdapterAlgorithm. 73
6.1 A ControlSignature. 81
6.2 An ExecutionSignature. 83
6.3 An InputDomain declaration. 83
6.4 An overloaded parenthesis operator of a worklet. 84
6.5 Declaration and use of a field map worklet. 85
7.1 Behavior of vtkm::cont::arg::TypeCheck. 88
7.2 Behavior of vtkm::cont::arg::Transport. 89
7.3 Declaring vtkm::internal::FunctionInterface. 91

7.4 Using vtkm::internal::make_FunctionInterface 92
7.5 Getting the arity of a FunctionInterface 92
7.6 Using FunctionInterface::GetParameter() 92
7.7 Using FunctionInterface::SetParameter() 93
7.8 Invoking a FunctionInterface 93
7.9 Invoking a FunctionInterface with a transform 94
7.10 Getting return value from FunctionInterface safely 94
7.11 Appending parameters to a FunctionInterface 95
7.12 Replacing parameters in a FunctionInterface 96
7.13 Chaining Replace and Append with a FunctionInterface 96
7.14 Using a static transform of function interface class 97
7.15 Using a dynamic transform of a function interface 98
7.16 Using DynamicTransform to cast dynamic arrays in a function interface 99
7.17 Using the ForEach feature of FunctionInterface 100

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 9), 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, export 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 interject 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.

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 * 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.

[The following example uses the POSIX math floor function. It should be replaced with the VTK-m version of floor when available.]

Example 2.8: Using `TypeTraits` for a generic remainder.

```
#include <vtkm/TypeTraits.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)
{
  T quotient = numerator / denominator;
  return (quotient - floor(quotient))*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 4.3).

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::Float64,3> > {  };

// 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 4.3). 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 4.3).

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

Provided Filters

[Write this once some filters are provided. Should also make a reference to using worklets (Chapter 6).]

Chapter 4

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.

4.1 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.

4.1.1 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.

[This is not currently the case in VTK-m, but should be once these adapters are implemented.]

- 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.
- 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.

[Most of these are not currently implemented.]

`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.

`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 4.1: 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.

4.1.2 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.

[Most of these are not currently implemented. The directories may change.]

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::cuda::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/cuda/cont/DeviceAdapterCuda.h.

vtkm::openmp::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.

vtkm::tbb::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/tbb/cont/DeviceAdapterTBB.h.

[Add example. Not enough implemented to make a worthwhile example yet.]

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 that 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 worklet.

[Add example when worklets and dispatchers are implemented.]

4.2 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.

4.2.1 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 4.2: 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 4.3: 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 4.4: 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 4.5: 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 4.5 where to be placed within a callable function or method, it could cause the vtkm::cont::ArrayHandle to become invalid.

Example 4.6: 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.
}
```

4.2.2 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 4.7: 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 4.8: 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 4.9: 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 `typedef`s 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 `typedef`s 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 4.10: 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;
    }
```

```
  }
}
```

4.2.3 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 4.1 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 4.8, but here is a quick example of using these execution array portals in a simple functor.

Example 4.11: 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.

4.2.4 Basic Storage

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

Example 4.12: 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 4.13: 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.

4.2.5 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 4.14: 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 4.2.2 and is generally straightforward for simple containers like this. Here is an example implementation for our `FooFieldsDeque` container.

Example 4.15: 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 4.16: 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 4.15.

Example 4.17: 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 4.18: 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:
  typedef vtkm::cont::ArrayHandle<float, StorageTagFooPressure>
      Superclass;

  VTKM_CONT_EXPORT
  ArrayHandleFooPressure(FooFieldsDeque *container)
    : Superclass(StorageType(container)) {  }
};
```

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.]

[Right now we don't have things that run in the execution environment that do anything useful, so update this example once that exists.]

Example 4.19: 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.
}
```

4.2.6 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 4.20: 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 4.21: 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 4.22: 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 4.23: Custom implicit array handle for even numbers.

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

class ArrayHandleDoubleIndex
```

```
    : public vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
{
  typedef vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
      Superclass;
public:
  VTKM_CONT_EXPORT
  ArrayHandleDoubleIndex(vtkm::Id numberOfValues)
    : Superclass(DoubleIndexFunctor(), numberOfValues) {  }
};
```

VTK-m comes with some examples of implicit storage. `vtkm::cont::ArrayHandleConstant` 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::ArrayHandleCounting` returns the index as the value with a possible offset. The counting array is 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.

4.2.7 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 4.24: 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 4.25: 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 4.26: 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:
  typedef typename ArrayHandleType::ValueType ValueType;

private:
  typedef vtkm::cont::ArrayHandleTransform<
      ValueType, ArrayHandleType, ScaleBiasFunctor<ValueType> > Superclass;

public:
  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);
}
```

4.2.8 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 4.27: 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 4.28: 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.

4.2.9 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 4.29: 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 4.30: Using make_ArrayHandleZip.

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

4.2.10 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 4.2.7, 4.2.8, and 4.2.9, 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).

[Technically there is a problem with this portal in that it calls the derived portals' Get and Set within methods exported in both control and execution. Portals that are not explicit tend to be exported in either one or the other, so these exports will fail under CUDA. In reality you need separate versions for control and execution environments.]

Example 4.31: 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_EXEC_CONT_EXPORT
  ArrayPortalConcatenate() : Portal1(), Portal2() {  }

  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).
  template<typename OtherP1, typename OtherP2>
  VTKM_EXEC_CONT_EXPORT
  ArrayPortalConcatenate(const ArrayPortalConcatenate<OtherP1,OtherP2> &src)
    : Portal1(src.GetPortal1()), Portal2(src.GetPortal2()) {  }

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

  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_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 4.32: Storage for derived container of concatenated arrays.

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

namespace vtkm {
namespace cont {
namespace internal {

template<typename T, typename Storage1, typename Storage2>
class Storage<
    T,
    StorageTagConcatenate<
      vtkm::cont::ArrayHandle<T, Storage1>,
      vtkm::cont::ArrayHandle<T, Storage2> > >
{
  typedef vtkm::cont::ArrayHandle<T, Storage1> ArrayHandleType1;
  typedef vtkm::cont::ArrayHandle<T, Storage2> ArrayHandleType2;

public:
  typedef T 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 4.33: 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 4.33, the prototype for the constructor must be equivalent to the following.

Example 4.34: 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 4.31 and 4.32, the following provides an ArrayTransfer appropriate for the derived storage.

Example 4.35: 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 4.36: ArrayHandle for derived storage of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
class ArrayHandleConcatenate
    : public vtkm::cont::ArrayHandle<
        typename ArrayHandleType1::ValueType,
        StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2> >
{
  typedef StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2>
      StorageTag;
  typedef vtkm::cont::ArrayHandle<
        typename ArrayHandleType1::ValueType, StorageTag>
      Superclass;
  typedef typename Superclass::ValueType ValueType;
  typedef vtkm::cont::internal::Storage<ValueType,StorageTag>
      StorageType;

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

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.

4.3 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 4.37: 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;
}
```

4.3.1 Querying and Casting

Data pointed to by a DynamicArrayHandle is not directly accessible. Before the data 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 `IsTypeAndStorage` method. This templated method takes an example component type and an example storage type as arguments and returns whether the underlying array matches both types.

Example 4.38: Querying the component and storage types of a DynamicArrayHandle.

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

```
// 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 types of the DynamicArrayHandle are known, it can be cast to a concrete ArrayHandle, which has access to the data as described in Section 4.2. This is done with the CastToArrayHandle method. This templated method takes an example component type and an example storage type as arguments and returns an ArrayHandle also pointing to the array. If the given types are incorrect, then CastToArrayHandle throws a vtkm::cont::-ErrorControlBadValue exception.

Example 4.39: Casting a DynamicArrayHandle to a concrete ArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32> concreteHandle =
    dynamicHandle.CastToArrayHandle(vtkm::Float32(),
                                    VTKM_DEFAULT_STORAGE_TAG());
```

4.3.2 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 4.40: 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++)
    {
      std::cout << " " << portal.Get(index);
    }
```

```
    std::cout << std::endl;
  }
};

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

4.3.3 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 4.40, 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 4.40 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 4.41: 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 4.42: Specifying a single component type in a `DynamicArrayHandle`.

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

Storage type lists can be changed similarly.

Example 4.43: Specifying different storage types in a `DynamicArrayHandle`.

```
struct MyIdStorageList :
    vtkm::ListTagBase<
        vtkm::cont::StorageTagBasic,
        vtkm::cont::ArrayHandleCounting<vtkm::Id>::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 4.44: Specifying both component and storage types in a `DynamicArrayHandle`.

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

4.4 Dynamic Point Coordinates

Point coordinates in VTK-m are represented as arrays much like any other field. In that aspect, a point coordinate array could be stored in a `DynamicArrayHandle`. However, point coordinates are often constructed as functional values or are derived and combined from other arrays. Constructing lists of storage types for `DynamicArrayHandle` is problematic.

Thus, VTK-m provides a similar class named `vtkm::cont::DynamicPointCoordinates` designed to hold point coordinates of an indeterminate type. `DynamicPointCoordinates` has an interface and behavior very similar to that for `DynamicArrayHandle` except instead of holding a reference to a `ArrayHandle` object, it holds a reference to a `PointCoordinates*` object. A point coordinate object holds enough state to reproduce point coordinates. VTK-m comes with the following point coordinate classes.

vtkm::cont::PointCoordinatesArray Holds point coordinates explicitly listed in an ArrayHandle. The component type of the array must be vtkm::Vec with three components, but the array can use any type of storage.

vtkm::cont::PointCoordinatesUniform Represents the point coordinates of a uniform rectilinear grid with a given extent, origin, and spacing.

A DynamicPointCoordinates object is created with one of these point coordinates objects. The following example demonstrates creating DynamicPointCoordinates with common point coordinate types.

Example 4.45: Creating DynamicPointCoordinates objects.

```
std::vector<vtkm::Vec<vtkm::Float32,3> > coordinatesBuffer;
// Fill coordinatesBuffer...
vtkm::cont::PointCoordinatesArray arrayPointCoords(
      vtkm::cont::make_ArrayHandle(coordinatesBuffer));
vtkm::cont::DynamicPointCoordinates pointCoords1(arrayPointCoords);

vtkm::cont::PointCoordinatesUniform uniformPointCoords(
      vtkm::Extent3(vtkm::Id3(0,0,0),vtkm::Id3(10,10,10)),
      vtkm::make_Vec(0, 0, 0),
      vtkm::make_Vec(1, 1, 1));
vtkm::cont::DynamicPointCoordinates pointCoords2(uniformPointCoords);
```

4.4.1 Querying and Casting

Given a DynamicPointCoordinates object, you can query the type of point coordinates it contains using the `IsPointCoordinateType` method. This templated method takes an example point coordinate type as an argument and returns whether the underlying array matches.

Example 4.46: Querying the point coordinate type of DynamicPointCoordinates.

```
vtkm::cont::PointCoordinatesUniform uniformPointCoords(
      vtkm::Extent3(vtkm::Id3(0,0,0),vtkm::Id3(10,10,10)),
      vtkm::make_Vec(0, 0, 0),
      vtkm::make_Vec(1, 1, 1));
vtkm::cont::DynamicPointCoordinates dynamicPointCoords(uniformPointCoords);

// This returns true
bool isUniform = dynamicPointCoords.IsPointCoordinateType(
                    vtkm::cont::PointCoordinatesUniform());

// This returns false
bool isArray = dynamicPointCoords.IsPointCoordinateType(
                  vtkm::cont::PointCoordinatesArray());
```

Once the types of the DynamicPointCoordinates object are known, it can be cast to a concrete point coordinates type with the `CastToPointCoordinates` method. This templated method takes an example point coordinate type as an argument and returns the concrete point coordinates object. If the given type is incorrect, then `CastToPointCoordinates` throws a vtkm::cont::ErrorControlBadValue exception.

Example 4.47: Casting DynamicPointCoordinates to concrete point coordinates.

```
vtkm::cont::PointCoordinatesUniform concretePointCoords =
    dynamicPointCoords.CastToPointCoordinates(
      vtkm::cont::PointCoordinatesUniform());
```

All this said, casting to a point coordinate types is generally not very useful. It is more common to use the following cast mechanism to cast all the way to an unknown array type.

4.4.2 Casting to Unknown Array Types

The use case of DynamicPointCoordinates is to cast to some form of ArrayHandle, which has a storage that is often complex and usually unknown. To perform this operation DynamicPointCoordinates has a method named CastAndCall that attempts to cast the point coordinates to an array type.

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 4.48: Operating on DynamicPointCoordinates with CastAndCall.

```
struct PrintPointCoordinatesFunctor
{
  template<typename ArrayHandleType>
  VTKM_CONT_EXPORT
  void operator()(const ArrayHandleType &array) const
  {
    typename ArrayHandleType::PortalConstControl portal =
          array.GetPortalConstControl();
    for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
    {
      std::cout << " " << portal.Get(index);
    }
    std::cout << std::endl;
  }
};

template<typename DynamicPointCoordinatesType>
void PrintPointCoordinates(const DynamicPointCoordinatesType &pointCoords)
{
  pointCoords.CastAndCall(PrintPointCoordinatesFunctor());
}
```

4.4.3 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.

Common type lists for point coordinates are defined in vtkm/cont/PointCoordinatesListTag.h. This header defines the list tag vtkm::cont::PointCoordiantesListTagCommon that

contains the common point coordinates types of PointCoordinatesArray and PointCoordinatesUniform. This header file also defines the single item lists vtkm::cont::PointCoordiantesListTagArray and vtkm::cont::PointCoordinatesListTagUniform.

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

The vtkm/cont/PointCoordinatesListTag.h header also defines a macro named VTKM_DEFAULT_POINT_COORDINATES_LIST_TAG that defines a default list of point coordinates to try in DynamicPointCoordinates. This list can be overridden by defining the VTKM_DEFAULT_POINT_COORDINATES_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.

DynamicPointCoordinates uses the aforementioned list to find an appropriate cast to a concrete point coordinates class. Once such a class is found, CastAndCall then delegates to this class the task of casting to an ArrayHandle. This delegate point coordinates class may (or may not) use a list of component types and storage types similar to those used in the CastAndCall of DynamicArrayHandle (as described in Section 4.3.3).

There is a form of CastAndCall that accepts tags for the list of point coordinate types, the list of component types and the list of 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 PrintPointCoordinates function in Example 4.48, passing these tags is inconvenient at best.

To address this use case, DynamicPointCoordinates has a set of methods named ResetPointCoordinatesList, ResetTypeList, and ResetStorageList. These methods return a new object with that behaves just like a DynamicPointCoordinates object with identical state except that the cast and call functionality uses the specified point coordinates type, component type, or storage type instead of the default. (Note that PrintPointCoordinates in Example 4.48 is templated on the type of DynamicPointCoordinates. This is to accommodate using the objects from the Reset*List methods, which have the same behavior but different type names.)

[TODO: Add some examples here. It would be helpful if there were more point coordinates classes (which I expect in the future) to make compelling examples. See the end of Section 4.3.3 for a list of related examples.]

4.4.4 Defining new Point Coordinate Types

Although VTK-m comes with a set of prescribed point coordinate types, it is also possible to define your own point coordinates class.

Defining a point coordinates class is very straightforward and only has two requirements.

First, a point coordinates class that publicly inherits from `vtkm::cont::internal::PointCoordinatesBase`. Second, a point coordinates class defines a const `CastAndCall` method that accepts three templated arguments: a functor, a list of component types, and a list of storage types. This method finds a concrete array class and calls the functor with this array. The list of component types and list of storage types may be used or may be ignored.

By convention, point coordinate class names begin with `PointCoordinates` and they have constructors that establish the state for the arrays they give as well as a default constructor.

The following example creates a point coordinates class that simply counts the vertex positions. [We can make a much more compelling exampling once transform arrays are implemented.]

Example 4.49: A custom point coordinates class.

```
class PointCoordinatesCounting :
    public vtkm::cont::internal::PointCoordinatesBase
{
public:
  VTKM_CONT_EXPORT
  PointCoordinatesCounting() : NumberOfPoints(0) {  }

  VTKM_CONT_EXPORT
  PointCoordinatesCounting(vtkm::Id numPoints) : NumberOfPoints(numPoints) {  }

  template<typename Functor, typename TypeList, typename StorageList>
  VTKM_CONT_EXPORT
  void CastAndCall(const Functor &f, TypeList, StorageList) const
  {
    typedef vtkm::Vec<vtkm::FloatDefault,3> PointCoordinatesType;

    vtkm::cont::ArrayHandleCounting<PointCoordinatesType>
        countingArray(PointCoordinatesType(0.0), this->NumberOfPoints);
    f(countingArray);
  }
private:
  vtkm::Id NumberOfPoints;
};
```

When creating a point coordinates class, it is prudent to also create list tags containing the new class so that it may be found when `CastAndCall` is invoked. Here is a simple example of creating lists for the previously defined point coordinates and using them.

Example 4.50: Using a custom point coordinates class.

```
struct PointCoordinatesListTagCounting :
    public vtkm::ListTagBase<PointCoordinatesCounting>
{  };

struct MyPointCoordinatesListTag :
    public vtkm::ListTagJoin<PointCoordinatesListTagCounting,
                             vtkm::cont::PointCoordinatesListTagCommon>
{  };

void Foo()
{
  PointCoordinatesCounting countingPoints(10);

  vtkm::cont::DynamicPointCoordinates dynamicPoints(countingPoints);

  PrintPointCoordinates(
```

```
        dynamicPoints.ResetPointCoordinatesList(MyPointCoordinatesListTag()));
}
```

4.5 Grid Structures

[Not implemented yet. These will be based on EAVL data structures. The name of this section might change.]

4.6 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 4.51: 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;
```

4.7 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 4.52: 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::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.

vtkm::cont::ErrorControlOutOfMemory Thrown when a VTK-m function or method tries to allocate an array and fails.

4.8 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 4.53: 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.

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.

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.

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. [This does not seem to be implemented right now, probably because we don't have the zip array handle, but it should be.]

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.

Chapter 5

Execution Environment

[Write this.]

Chapter 6

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.

6.1 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 6.4.5. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 7.

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.

6.2 Dispatchers

Worklets, both those provided by VTK-m as listed in Section 6.3 and ones created by a user as described in Section 6.4, 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 6.3. Also, for any worklet, the `Invoke` arguments can be gleaned from the control signature, which is described in Section 6.4.1.

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 6.1. Many examples of using these dispatchers are provided in Section 6.3.

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

6.3 Provided Worklets

[Write this once some worklets are provided.]

6.4 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 must contain an overload of the parenthesis operator, which is the method that is executed in the execution environment.

5. The class must publicly inherit from a base worklet class that specifies the type of operation being performed.

[Perhaps have an annotated example of a worklet declaration like the one in Figure 3.3 of the Dax report.]

6.4.1 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 6.1: 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 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 6.4.5.

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.

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`.

The signature tags and their modifiers are described in greater detail in the following section on worklet types.

6.4.2 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 6.4.4) in terms of arity and argument semantics.

Example 6.2: 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.

6.4.3 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 `typedef`ed 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 6.3: An `InputDomain` declaration.

```
typedef _1 InputDomain;
```

6.4.4 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.

[It might be better to have an example with a more straightforward template.]

Example 6.4: An overloaded parenthesis operator of a worklet.

```
template<typename T, vtkm::IdComponent Size>
VTKM_EXEC_EXPORT
vtkm::Float64 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.

6.4.5 Worklet Types

There are multiple worklet types provided by VTK-m, each designed to support a particular type of operation. This section will define each of the worklet types, identify the generic superclass that a worklet instance should derive, identify the signature tags and their meanings, and give an example of the worklet in use.

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 its `ControlSignature`:

`FieldIn` An input field. This tag corresponds to a `vtkm::cont::ArrayHandle` passed to `Invoke`, and each invocation of the worklet gets a single value in this array. Dynamic

classes like vtkm::cont::DynamicArrayHandle and vtkm::cont::DynamicPointCoordinates that are implicitly cast to ArrayHandles are also supported. FieldIn is templated with a single argument that specifies the types that can be in an array. Any type list tag can be used for this argument, but the most common entries are described in Section 6.4.1 starting on page 81.

FieldOut An output field. This tag corresponds to a vtkm::cont::ArrayHandle or vtkm::cont::DynamicArrayHandle passed to Invoke. The array is resized appropriately, and each invocation of the worklet sets a single value in this array. FieldOut is templated with a single argument that specifies the types that can be in an array. Any type list tag can be used for this argument, but the most common entries are described in Section 6.4.1 starting on page 81.

ExecObject An execution object, which is passed directly from the control environment to the execution environment and then passed to each invocation of the worklet. The execution object must derive vtkm::exec::ExecutionObjectBase.

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

_1, _2, _3,... The numeric tags point to arguments in the ControlSignature. The data passed to the worklet come from this identified argument.

WorkIndex A unique identifier for the invocation instance of the worklet. The work index corresponds to the index of the domain.

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

[When VTK-m math library is available, use it in this example.]

Example 6.5: Declaration 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>

// TODO: Use VTK-m math when available.
#include <math.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
```

```
  vtkm::Float64 operator()(const vtkm::Vec<T,Size> &inVector) const
  {
    T magnitudeSquared = vtkm::dot(inVector, inVector);
    // TODO: Use VTK-m math when available.
    return sqrt(vtkm::Float64(magnitudeSquared));
  }
};

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);
}
```

Chapter 7

Advanced Worklet Customization

Chapter 6 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.

7.1 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.]

7.1.1 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 7.1: 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
}
```

7.1.2 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 7.2: 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);
}
```

7.1.3 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.

7.2 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.

7.2.1 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 7.3: 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 7.4: Using vtkm::internal::make_FunctionInterface.

```
const char *s = "Hello World";
static const size_t BUFFER_SIZE = 100;
char buffer[BUFFER_SIZE];

strlenInterface =
    vtkm::internal::make_FunctionInterface<size_t>(s);

strncpyInterface =
    vtkm::internal::make_FunctionInterface<char *>(buffer, s, BUFFER_SIZE);
```

7.2.2 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 7.5: 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 7.6: Using FunctionInterface::`GetParameter()`.

```
void GetFirstParameterResolved(
    const vtkm::internal::FunctionInterface<void(const char *)> &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 7.7: Using `FunctionInterface::SetParameter()`.

```
void SetFirstParameterResolved(
    vtkm::internal::FunctionInterface<void(const char *)> &interface,
    const char *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>());
}
```

7.2.3 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 7.8: Invoking a `FunctionInterface`.

```
  vtkm::internal::FunctionInterface<size_t(const char *)> strlenInterface =
      vtkm::internal::make_FunctionInterface<size_t>("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 7.9: 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>("42",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::FunctionInterfaceReturnContainer. 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 7.10: 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());
}
```

7.2.4 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 7.11: Appending parameters to a `FunctionInterface`.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

typedef FunctionInterface<void(const char *, vtkm::Id)>
    InitialFunctionInterfaceType;
InitialFunctionInterfaceType initialFunctionInterface =
    make_FunctionInterface<void>("Hello World", vtkm::Id(42));

typedef FunctionInterface<void(const char *, vtkm::Id, const char *)>
    AppendedFunctionInterfaceType1;
AppendedFunctionInterfaceType1 appendedFunctionInterface1 =
    initialFunctionInterface.Append("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 7.12: Replacing parameters in a `FunctionInterface`.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

typedef FunctionInterface<void(const char *, vtkm::Id)>
    InitialFunctionInterfaceType;
InitialFunctionInterfaceType initialFunctionInterface =
    make_FunctionInterface<void>("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, const char *>::type
    ReplacedFunctionInterfaceType2;
ReplacedFunctionInterfaceType2 replacedFunctionInterface2 =
    initialFunctionInterface.Replace<2>("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 7.13: 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;
```

```
}
```

7.2.5 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 7.14: 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 7.15: 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 7.16: 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>());
}
```

7.2.6 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 7.17: 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;
}
```

7.3 Invocation Objects

7.4 Creating New `ControlSignature` Tags

7.5 Creating New `ExecutionSignature` Tags

7.6 Creating New Worklet Types

7.6.1 New Worklet Superclasses

7.6.2 Dispatch Workflow

7.6.3 New Dispatch Classes

Chapter 8

OpenGL Interoperability

Chapter 9

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

_1, 83, 85
_2, 83, 85
_3, 85

algorithm, 73–75
AllTypes, 81
arg namespace, 88–90
arity, 92
ArrayHandle, 10, 11, 18, 38, 39, 42, 44, 45, 48, 62, 84, 85, 88, 89
ArrayHandle.h, 18, 106
ArrayHandleCompositeVector, 62
ArrayHandleConstant, 51
ArrayHandleCounting, 51
ArrayHandleImplicit, 49, 50
ArrayHandleImplicit.h, 50
ArrayHandlePermutation, 53
ArrayHandlePermutation.h, 54
ArrayHandleTransform, 51, 52
ArrayHandleUniformPointCoordinates, 51
ArrayHandleZip, 54
ArrayHandleZip.h, 55
ArrayPortalFromIterators, 40
ArrayPortalToIteratorBegin, 42
ArrayPortalToIteratorEnd, 42
ArrayPortalToIterators, 41
ArrayPortalToIterators.h, 42
ArrayTransfer, 11, 58, 59
array handle, 38–62
    adapting, 45–49
    derived, 55–62
    dynamic, 63–66
    fancy, 49–62
    implicit, 49–51
    permutation, 53–54
    storage, 44–62
    transform, 51–53
    zip, 54–55
array portal, 40–43
array transfer, 58–62
aspect, 90–91
    default, 90
    work index, 91
AspectTagDefault, 90
AspectTagWorkIndex, 91
assert, 72
Assert.h, 72

CellField, 108
CMake configuration
    VTKM_FORCE_ANSI, 108
    VTKM_USE_64BIT_IDS, 20
    VTKM_USE_DOUBLE_PRECISION, 20
CommonTypes, 82
ComponentType, 25
ConfigureFor32.h, 20
ConfigureFor64.h, 20
constant export, 19
cont namespace, 17, 18
control signature, 7, 12, 80, 81, 83–85, 87, 91, 101
control environment, 17, 35–75
control signature, 81–82
copy, 73
CUDA, 19, 36, 37
cuda namespace, 18

derived storage, 55–62
detail namespace, 18
DeviceAdapter.h, 35
DeviceAdapterAlgorithm, 12, 43, 73
DeviceAdapterCuda.h, 37
DeviceAdapterOpenMP.h, 37
DeviceAdapterSerial.h, 37
DeviceAdapterTagCuda, 37
DeviceAdapterTagOpenMP, 37
DeviceAdapterTagSerial, 37
DeviceAdapterTagTBB, 37
DeviceAdapterTBB.h, 37
device adapter, 35–38, 73–75
    algorithm, 73–75
device adapter tag, 35, 37–38
DimensionalityTag, 24
dispatcher, 80

DispatcherMapField, 80, 84
dot, 21
DynamicArrayHandle, 30, 63, 85, 99
DynamicPointCoordinates, 66, 85, 99
DynamicTransform, 99
dynamic array handle, 63–66
    cast, 64–66
    construct, 63
    query, 63
dynamic point coordinates, 66–71
    cast, 67–69
    construct, 67
    query, 67

environments, 17
Error, 72
ErrorControl, 72
ErrorControlAssert, 72
ErrorControlBadValue, 60, 64, 67, 72
ErrorControlInternal, 60, 72
ErrorControlOutOfMemory, 73
ErrorExecution, 72, 74
errors, 72–73
    assert, 72
    execution environment, 72, 74
exec namespace, 17, 18
ExecObject, 85
execution signature, 7, 12, 81, 83–85, 87, 101
ExecutionObjectBase, 85, 88
ExecutionTypes, 42
execution environment, 17
execution signature, 83
export
    constant, 19
    control, 19, 84, 109
    execution, 19, 84, 109
Extent, 22
Extent.h, 22
Extent2, 22
Extent3, 22, 23
ExtentCellDimensions, 22
ExtentCellFlatIndexToTopologyIndex, 22
ExtentCellTopologyIndexToFlatIndex, 22
ExtentFirstPointOnCell, 23
ExtentNumberOfCells, 22
ExtentNumberOfPoints, 22
ExtentPointDimensions, 22
ExtentPointFlatIndexToTopologyIndex, 22
ExtentPointTopologyIndexToFlatIndex, 22

fancy array handle, 49–62
Fetch, 90
fetch, 90–91
    aspect, *see* aspect
    direct input array, 90
    direct output array, 90
    execution object, 91
FetchTagArrayDirectIn, 90
FetchTagArrayDirectOut, 90
FetchTagExecObject, 91
FieldCommon, 82
FieldIn, 84, 85
FieldOut, 85
FieldPointIn, 81
field map worklet, 79, 84–86
Float32, 10, 19, 24, 29, 82, 108
Float64, 19, 29, 82, 108
FloatDefault, 20, 109
function export, 19, 84, 109
functional array, 49–51
FunctionInterface, 12, 91
FunctionInterfaceReturnContainer, 94
function interface, 91–100
    append parameter, 95–96
    dynamic transform, 98–100
    for each, 100
    invoke, 93–94
    replace parameter, 96
    static transform, 97–98
function signature, 91
functor, 50
FunctorBase, 74

GetComponent, 26
GetPortalConstControl, 42
GetPortalControl, 42

HasMultipleComponents, 26

Id, 20, 21, 25, 29, 50, 60, 73–75, 82, 108
Id2, 21, 22, 29, 82

Id2Type, 82
Id3, 10, 21, 22, 25, 26, 29, 74, 82
Id3Type, 82
IdComponent, 20, 90, 97, 108
IdType, 82
implicit array handle, 49–51
implicit storage, 49–51
Index, 82
IndexTag, 92, 97, 98
input domain, 83
input domain, 12, 81, 83
Int16, 19
Int32, 19
Int64, 19
Int8, 19
Intel Threading Building Blocks, 36, 37
internal namespace, 18, 91
interoperability, 18
invoke, 80

less, 26
ListForEach, 30
lists, 27–31
    point coordinates, 68–69
    storage, 65
    types, 29–30
ListTag.h, 28, 30
ListTagBase, 28
ListTagEmpty, 28
ListTagJoin, 28
lower bounds, 73

make_ArrayHandle, 39
make_ArrayHandleImplicit, 50
make_ArrayHandlePermutation, 54
make_ArrayHandleTransform, 52
make_ArrayHandleZip, 55
make_FunctionInterface, 13, 92
make_Pair, 23
make_Vec, 20
map, 79
map field, *see* field map worklet
math namespace, 18
Matrix, 108
metaprogramming, 27
method export, 19, 84, 109

namespace, 17
    detail, 18
    internal, 18
    vtkm, 17, 91
    vtkm::cont, 17, 18
    vtkm::cont::arg, 88, 89
    vtkm::cuda, 18
    vtkm::exec, 17, 18
    vtkm::exec::arg, 90
    vtkm::internal, 91
    vtkm::math, 18
    vtkm::opengl, 18
    vtkm::openmp, 18
    vtkm::tbb, 18
    vtkm::worklet, 17, 18
NUM_COMPONENTS, 21, 25
NumericTag, 24

OpenGL, 18
opengl namespace, 18
OpenMP, 36, 37
openmp namespace, 18

packages, *see also* namespace, 17–18
Pair, 23, 54
permuted array handle, 53–54
point coordinates lists, 68–69
PointCoordiantesListTagArray, 69
PointCoordiantesListTagCommon, 68
PointCoordinatesArray, 67
PointCoordinatesBase, 70
PointCoordinatesListTag.h, 68, 69
PointCoordinatesListTagUniform, 69
PointCoordinatesUniform, 67
point coordinates
    dynamic, 66–71
PortalConstControl, 42
PortalControl, 42
PrepareForInPlace, 43
PrepareForInput, 43
PrepareForOutput, 43

RaiseError, 74

Scalar, 82

ScalarAll, 82
scan
    exclusive, 74
    inclusive, 73
schedule, 74
serial, 36, 37
SetComponent, 26
signature, 91
    control, 7, 12, 80–85, 87, 91, 101
    execution, 7, 12, 81, 83–85, 87, 101
signature tags, 81
    _1, 83, 85
    _2, 83, 85
    _3, 85
    AllTypes, 81
    CommonTypes, 82
    ExecObject, 85
    FieldCommon, 82
    FieldIn, 84, 85
    FieldOut, 85
    FieldPointIn, 81
    Id2Type, 82
    Id3Type, 82
    IdType, 82
    Index, 82
    Scalar, 82
    ScalarAll, 82
    Vec2, 82
    Vec3, 82
    Vec4, 82
    VecAll, 82
    VecCommon, 82
    WorkIndex, 83, 85, 91
sort, 74
    by key, 74
Storage, 11, 46, 47, 56, 59
storage, 44–62
    adapting, 45–49
    derived, 55–62
    implicit, 49–51
storage lists, 65
StorageBasic.h, 45
StorageListTag.h, 65
StorageListTagBasic, 65
StorageTagBasic, 45
stream compact, 74
synchronize, 75

tag, 24
    device adapter, 35, 37–38
    dimensionality, 24
    lists, 27–31
    multiple components, 26
    numeric, 24
    point coordinates lists, 68–69
    single component, 26
    storage lists, 65
    type lists, 29–30
    type traits, 24–25
    vector traits, 25–26
TBB, 36, 37
tbb namespace, 18
template metaprogramming, 27
Timer, 12, 71
timer, 71–72
ToVec, 26
traits, 24–27
transformed array, 51–53
Transport, 12, 88, 89
transport, 88–90
    execution object, 89
    input array, 89
    output array, 89
TransportTagArrayIn, 89
TransportTagArrayOut, 89
TransportTagExecObject, 89
type lists, 29–30
TypeCheck, 12, 87, 88
TypeCheckTagArray, 88
TypeCheckTagExecObject, 88
TypeListTag.h, 29, 30, 65
TypeListTagAll, 29
TypeListTagCommon, 30, 82
TypeListTagField, 29, 82
TypeListTagFieldScalar, 29, 82
TypeListTagFieldVec2, 29, 82
TypeListTagFieldVec3, 29, 82
TypeListTagFieldVec4, 29, 82
TypeListTagId, 29, 82
TypeListTagId2, 29, 82

TypeListTagId3, 29, 82
TypeListTagIndex, 29, 82
TypeListTagScalarAll, 29, 82
TypeListTagVecAll, 29, 82
TypeListTagVecCommon, 29, 82
Types.h, 19, 29, 107
TypeTraits, 10, 24
TypeTraitsIntegerTag, 24
TypeTraitsRealTag, 24
TypeTraitsScalarTag, 24
TypeTraitsVectorTag, 24
type check, 87–88
    array, 88
    execution object, 88

UInt16, 19
UInt32, 19
UInt64, 19
UInt8, 19
unique, 75
upper bounds, 75

Vec, 10, 20, 21, 26, 29, 67, 82, 107, 108
Vec2, 82
Vec3, 82
Vec4, 82
VecAll, 82
VecCommon, 82
VecTraits, 10, 25, 26
VecTraitsTagMultipleComponents, 26
VecTraitsTagSingleComponent, 26
vtkm namespace, 17, 91
vtkm/cont/ArrayHandle.h, 18, 106
vtkm/cont/ArrayHandleImplicit.h, 50
vtkm/cont/ArrayHandlePermutation.h, 54
vtkm/cont/ArrayHandleZip.h, 55
vtkm/cont/ArrayPortalToIterators.h, 42
vtkm/cont/Assert.h, 72
vtkm/cont/DeviceAdapter.h, 35
vtkm/cont/DeviceAdapterSerial.h, 37
vtkm/cont/PointCoordinatesListTag.h, 68, 69
vtkm/cont/StorageBasic.h, 45
vtkm/cont/StorageListTag.h, 65
vtkm/cuda/cont/DeviceAdapterCuda.h, 37
vtkm/internal/ConfigureFor32.h, 20
vtkm/internal/ConfigureFor64.h, 20
vtkm/openmp/cont/-DeviceAdapterOpenMP.h, 37
vtkm/tbb/cont/DeviceAdapterTBB.h, 37
vtkm::cont, 17, 18
vtkm::cont::arg, 88, 89
vtkm::cuda, 18
vtkm::exec, 17, 18
vtkm::exec::arg, 90
vtkm::internal, 91
vtkm::math, 18
vtkm::opengl, 18
vtkm::openmp, 18
vtkm::tbb, 18
vtkm::worklet, 17, 18
VTKM_ASSERT_CONT, 72
VTKM_CONT_EXPORT, 19, 109
VTKM_DEFAULT_DEVICE_ADAPTER_-TAG, 38
VTKM_DEFAULT_POINT_-COORDINATES_LIST_TAG, 69
VTKM_DEFAULT_STORAGE_LIST_TAG, 65
VTKM_DEFAULT_STORAGE_TAG, 45
VTKM_DEFAULT_TYPE_LIST_TAG, 30, 65
VTKM_DEVICE_ADAPTER, 36, 37
VTKM_DEVICE_ADAPTER_CUDA, 36
VTKM_DEVICE_ADAPTER_ERROR, 37
VTKM_DEVICE_ADAPTER_OPENMP, 36
VTKM_DEVICE_ADAPTER_SERIAL, 36
VTKM_DEVICE_ADAPTER_TBB, 36
VTKM_EXEC_CONSTANT_EXPORT, 19
VTKM_EXEC_CONT_EXPORT, 19, 84, 109
VTKM_EXEC_EXPORT, 19, 84, 109
VTKM_FORCE_ANSI, 108
VTKM_MAX_BASE_LIST, 28
VTKM_NO_64BIT_IDS, 20
VTKM_NO_DOUBLE_PRECISION, 20
VTKM_STORAGE, 44, 45
VTKM_STORAGE_BASIC, 45

VTKM_USE_64BIT_IDS, 20
VTKM_USE_DOUBLE_PRECISION, 20
vtkm/Extent.h, 22
vtkm/ListTag.h, 28, 30
vtkm/TypeListTag.h, 29, 30, 65
vtkm/Types.h, 19, 29, 107

WorkIndex, 83, 85, 91
worklet, 79–86
worklet namespace, 17, 18
WorkletMapField, 79, 80, 84
worklet types, 79, 84–86
    field map, 79, 84–86

zipped array handles, 54–55

DISTRIBUTION:

1	MS 1326	Kenneth Moreland, 1461
1	MS 1327	Ron Oldfield, 01461
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories