2008-7717P

Teuchos Utility Classes for Safer Memory Management in C++

Roscoe A. Bartlett
Department of Optimization & Uncertainty Estimation

Sandia National Laboratories

Trilinos Users Group Meeting, October 22, 2008

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Sandia National Laboratories

Current State of Memory Management in Trilinos C++ Code

- The Teuchos reference-counted pointer (RCP) class is being widely used
  - Memory leaks are becoming less frequent (but are not completely gone => circular references!)
  - Fewer segfaults from uninitailized pointers and accessing deleted objects …
- However, we still have problems …
  - Segfaults from improper usage of arrays of memory (e.g. off-by-one errors etc.)
  - Improper use of other types of data structures
- The core problem? => Ubiquitous high-level use of raw C++ pointers in our application (algorithm) code!
- What I am going to address in this presentation:
  - Adding new Teuchos utility classes similar to Teuchos::RCP to encapsulate usage of raw C++ pointers for:
    - handling of single objects
    - handling of contiguous arrays of objects
  - New Teuchos utility classes without reference counting to eliminate all raw pointers

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Outline

- Background
  - Problems with using raw C++ pointers at the application programming level
- Overview of Teuchos Memory Management Utility Classes
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Problems with using Raw Pointers at the Application Level

- The C/C++ Pointer:

```
Type *ptr;
```

- Problems with C/C++ Pointers
  - No default initialization to null => Leads to segfaults

```
int *ptr;
ptr[20] = 5; // BANG!
```

  - Using to handle memory of single objects

```
int *ptr = new int;
// No good can ever come of:
ptr++, ptr--, ++ptr, --ptr, ptr+i, ptr-i, ptr[i]
```

  - Using to handle arrays of memory:

```
int *ptr = new int[n];
// These are totally unchecked:
*(ptr++), *(ptr--), ptr[i]
```

  - Creates memory leaks when exceptions are thrown:

```
int *ptr = new int;
functionThatThrows(ptr);
delete ptr; // Will never be called if above function throws!
```

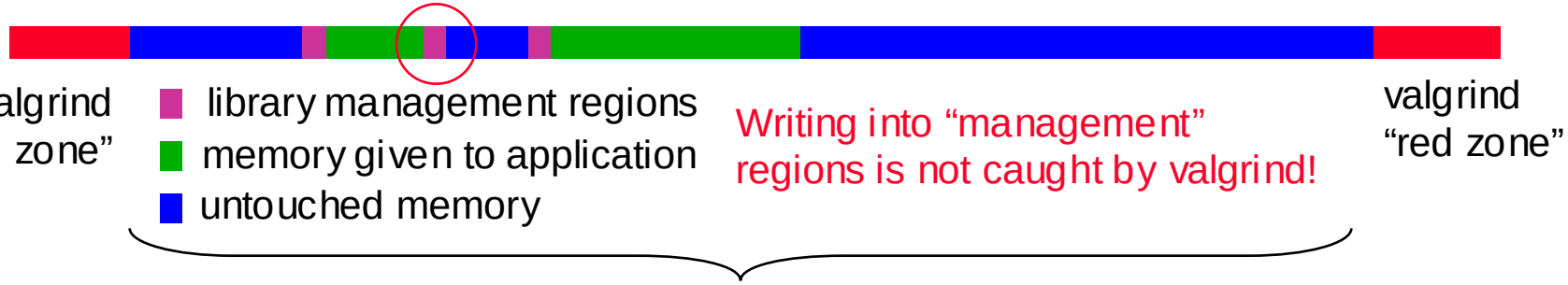
- How do we fix this?
  - Memory leaks? => Reference-counted smart pointers (not a 100% guarantee)
  - Segfaults? => Memory checkers like Valgrind and Purify? (far from a 100% guarantee)

Ineffectiveness of Memory Checking Utilities

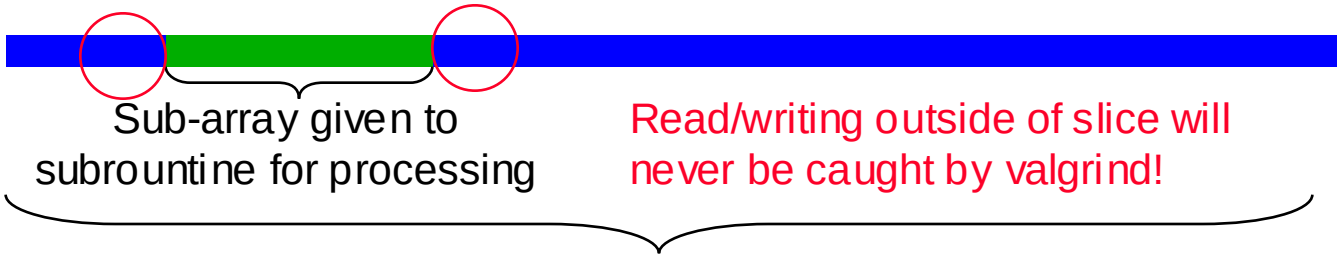
- Memory checkers like Valgrind and Purify only know about stack and heap memory requested from the system!

  => Memory managed by the library or the user program is totally unchecked

- Examples:
  - Library managed memory (e.g. GNU STL allocator)



  - Program managed memory



Memory checkers can never sufficiently verify your program!

Sandia National Laboratories

What is the Proper Role of Raw C++ Pointers?

AVOID USING RAW POINTERS AT THE APPLICATION PROGRAMMING LEVEL!

If we can't use raw pointers at the application level, then how can we use them?

- Basic mechanism for communicating with the compiler
- Extremely well-encapsulated, low-level, high-performance algorithms
- Compatibility with other software (again, at a very low, well-encapsulated level)

For everything else, let's use (existing and new) classes to more safely encapsulate our usage of memory!

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Basic Strategy for Safer "Pointer Free" Memory Usage

- Encapsulate raw pointers in specialized utility classes
  - In a debug build (--enable-teuchos-debug), all access to memory is checked at runtime … Maximize runtime checking and safety!
  - In an optimized build (default), no checks are performed giving raw pointer performance … Minimize/eliminate overhead!
- Define a different utility class for each major type of use case:
  - Single objects (persisting and non-persisting associations)
  - Views of arrays (persisting and non-persisting associations)
  - Containers (arrays, maps, lists, etc.)
  - etc …
- Allocate all objects in a safe way (i.e. don't call new directly at the application level!)
  - Use non-member constructor functions that return safe wrapped objects (See SAND2007-4078)
- Pass around encapsulated pointer(s) to memory using safe (checked) conversions between safe utility class objects

Definitions:

- Non-persisting association: Association that only exists within a single function call
- Persisting association: Association that exists beyond a single function call and where some "memory" of the object persists

Sandia National Laboratories

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Utility Classes for Memory Management of Single Classes

- Teuchos::RCP (Long existing class, first developed in 1997!)

  ```
  RCP<T> p;
  ```

  - Smart pointer class (e.g. usage looks and feels like a raw pointer)
  - Uses reference counting to decide when to delete object
  - Used for persisting associations with single objects
  - Allows for 100% flexibility for how object gets allocated and deallocated
  - [New] General approach for dealing with circular references
  - In a debug build, throws on dereferences of null and dangling references
  - Counterpart to boost::shared_ptr and std::tr1::shared_ptr

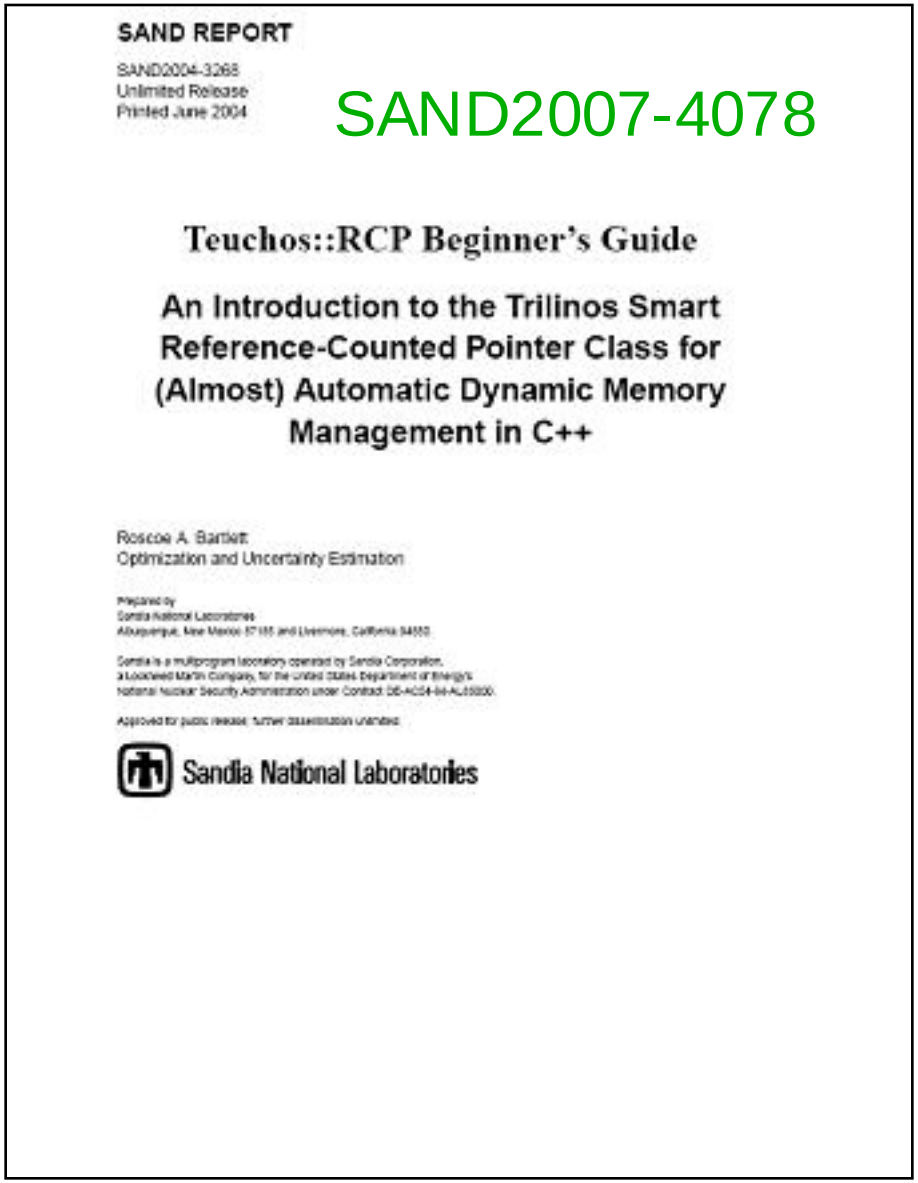
- Teuchos::Ptr (New class)

  ```
  void foo( const Ptr<T> &p );
  ```

  - Smart pointer class (e.g. operator->() and operator*())
  - Light-weight replacement for raw pointer T* to a single object
  - Default constructs to null
  - No reference counting! Used only for non-persisting association function arguments
  - In a debug build, throws on dereferences of null and dangling references
  - Integrated with other memory utility classes
  - No counterpart to boost or C++0x

Sandia National Laboratories

Teuchos::RCP Technical Report



SAND REPORT

SAND2004-3268
Unlimited Release
Printed June 2004

SAND2007-4078

Teuchos::RCP Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization and Uncertainty Estimation

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

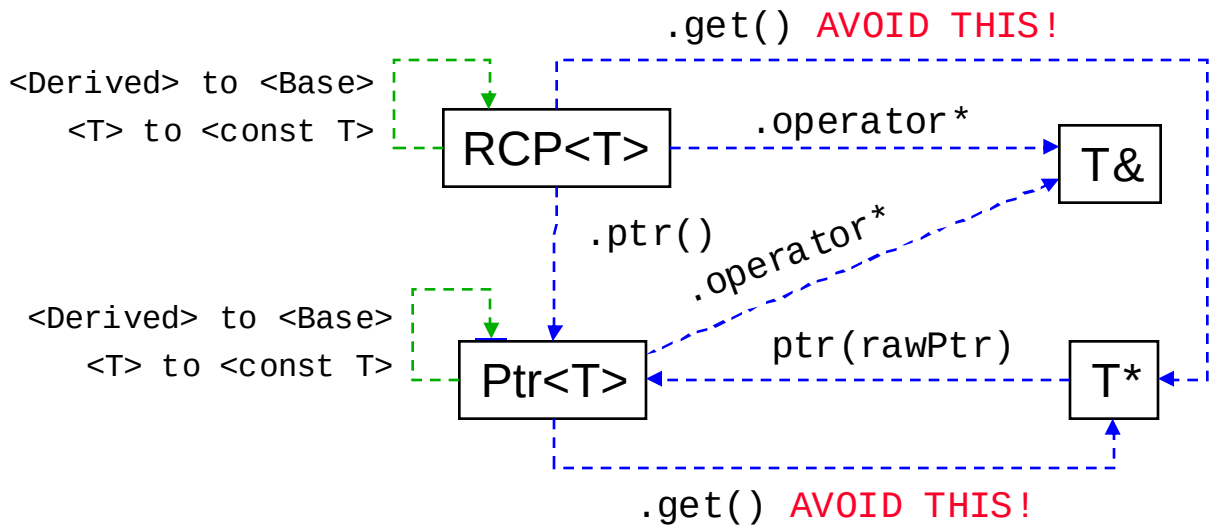
Approved for public release; further dissemination unlimited.

Sandia National Laboratories

http://trilinos.sandia.gov/documentation.html

Sandia National Laboratories

Conversions Between Single-Object Memory Management Types



Legend

<<implicit conversion>>

<<explicit conversion>>

Sandia National Laboratories

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Utility Classes for Memory Management of Arrays of Objects

- Teuchos::ArrayView (New class) => No equivenent in boost or C++0x

```
void foo( const ArrayView<T> &v );
```

  - Used to replace raw pointers as function arguments to pass arrays
  - Used for non-persisting associations only (i.e. only function arguments)
  - Allows for 100% flexibility for how memory gets allocated and sliced up
  - Minimal overhead in an optimized build, just a raw pointer and an integer

- Teuchos::ArrayRCP (Failry new class) => Counterpart to boost::array_ptr

```
ArrayRCP<T> v;
```

  - Used for persisting associations with fixed size arrays
  - Allows for 100% flexibility for how memory gets allocated and sliced up
  - Uses same reference-counting machinery as Teuchos::RCP
  - Gives up (sub)views as Teuchos::ArrayView and Teuchos::ArrayRCP objects

- Teuchos::Array (Existing class but majorly reworked)

```
Array<T> v;
```

  - A general purpose container class like std::vector (actually uses std::vector within)
  - All usage is runtime checked in a debug build
  - Gives up (sub)views as Teuchos::ArrayView objects

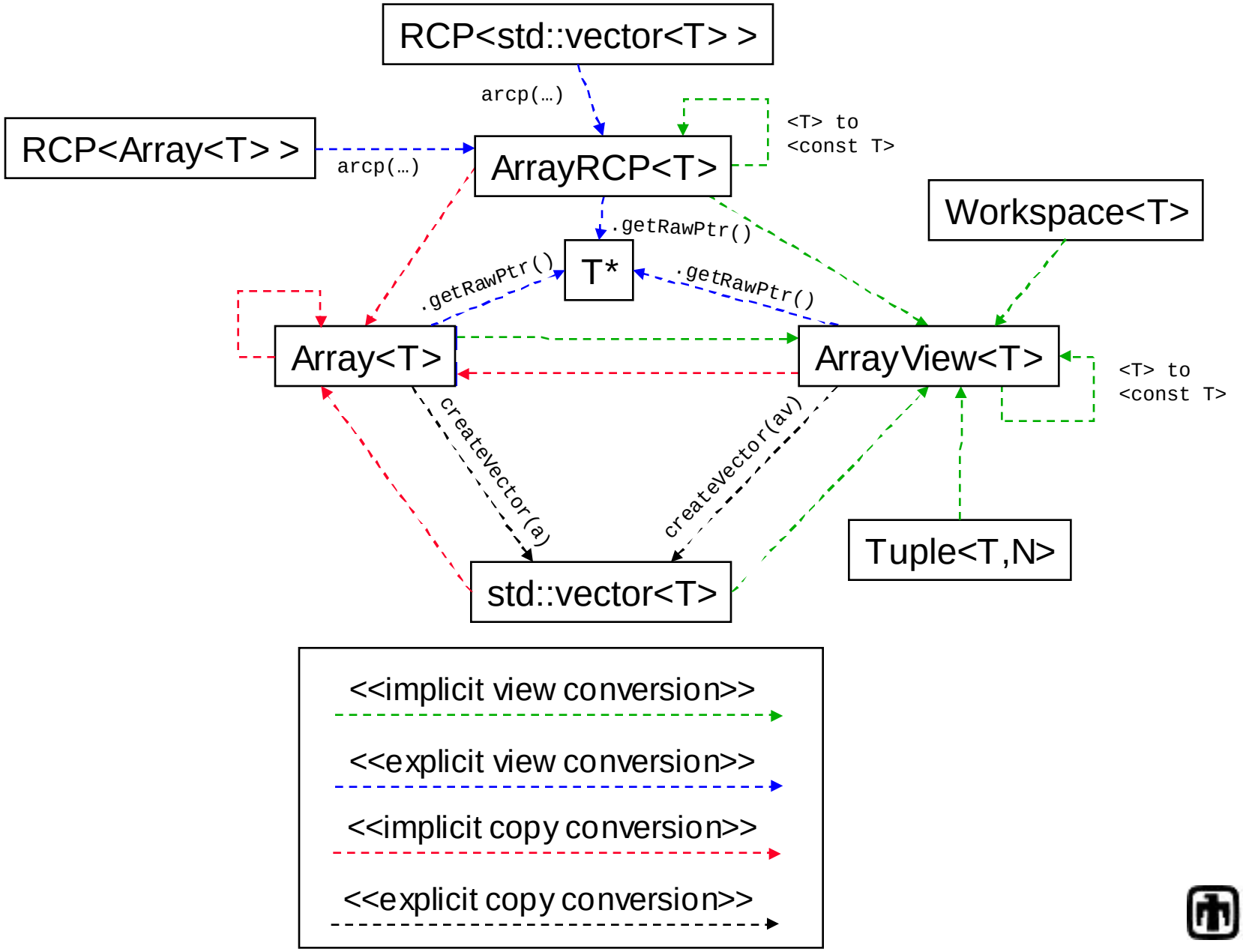
- Teuchos::Tuple (New class) => Counterpart to boost::array

```
Tuple<T,N> t;
```

  - Statically sized array class (replacement for built-in T[N])
  - Gives up (sub)views as Teuchos::ArrayView objects

Sandia National Laboratories

Conversions Between Array Memory Management Types



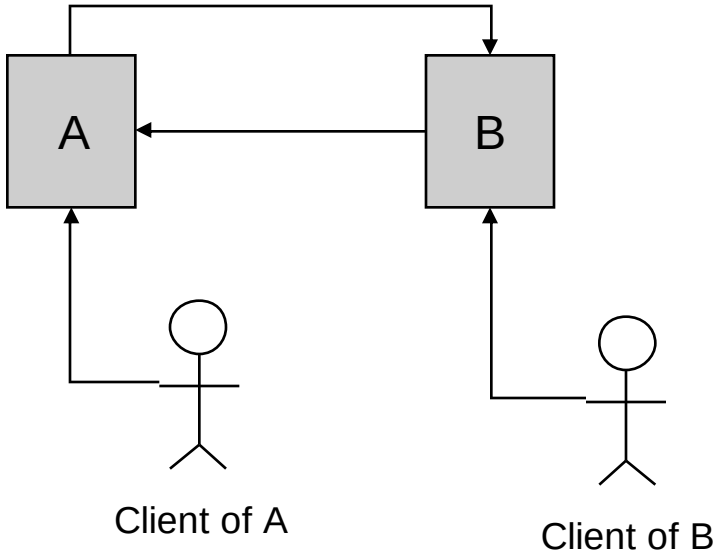
Sandia National Laboratories

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Handling of Circular References



- Without special handling, this will create memory leak of ‘A’ and ‘B’ objects
- Debugging circular references:
  - Teuchos::setTracingActiveRCPNodes(true);

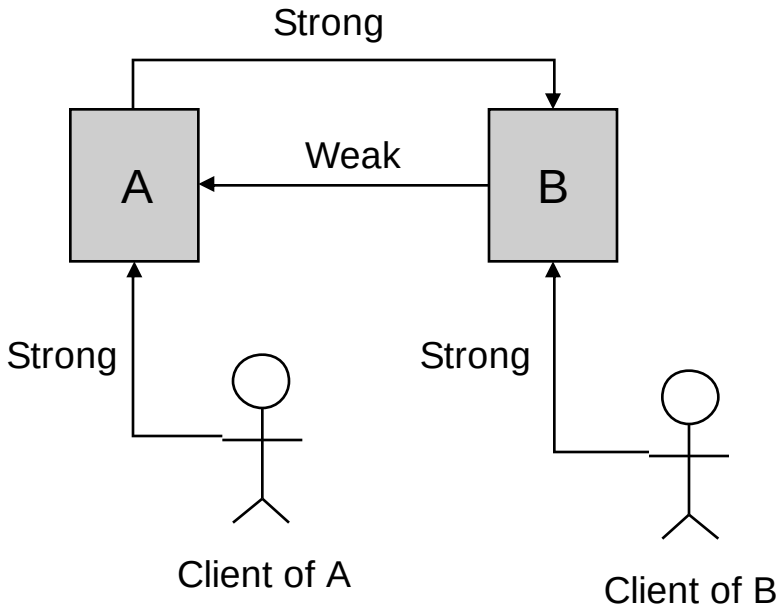
```
***
*** Warning! The following Teuchos::RCPNode objects were created but have
*** not been destroyed yet.  This may be an indication that these objects may
*** be involved in a circular dependency!  A memory checking tool may complain
*** that these objects are not destroyed correctly.
***

  RCPNode address = '0x890bb0', information = {T=‘C',Concrete T=‘C',p=0x890ee8,has_ownership=1}, call number = 1
  RCPNode address = '0x890be0', information = {T=‘A',Concrete T=‘A',p=0x890b90,has_ownership=1}, call number = 0
```

See: Trilinos/packages/teuchos/test/MemoryManagement/RCP_test.cpp

Sandia National Laboratories

Handling of Circular References



- New Feature: Strong/Weak RCPs
- Individual objects just store RCPs to shared objects
- Higher-level objects (i.e. factories) assign "strength"
  - Example: We know that 'B' will not access 'A' after 'A' gets deleted!

See:

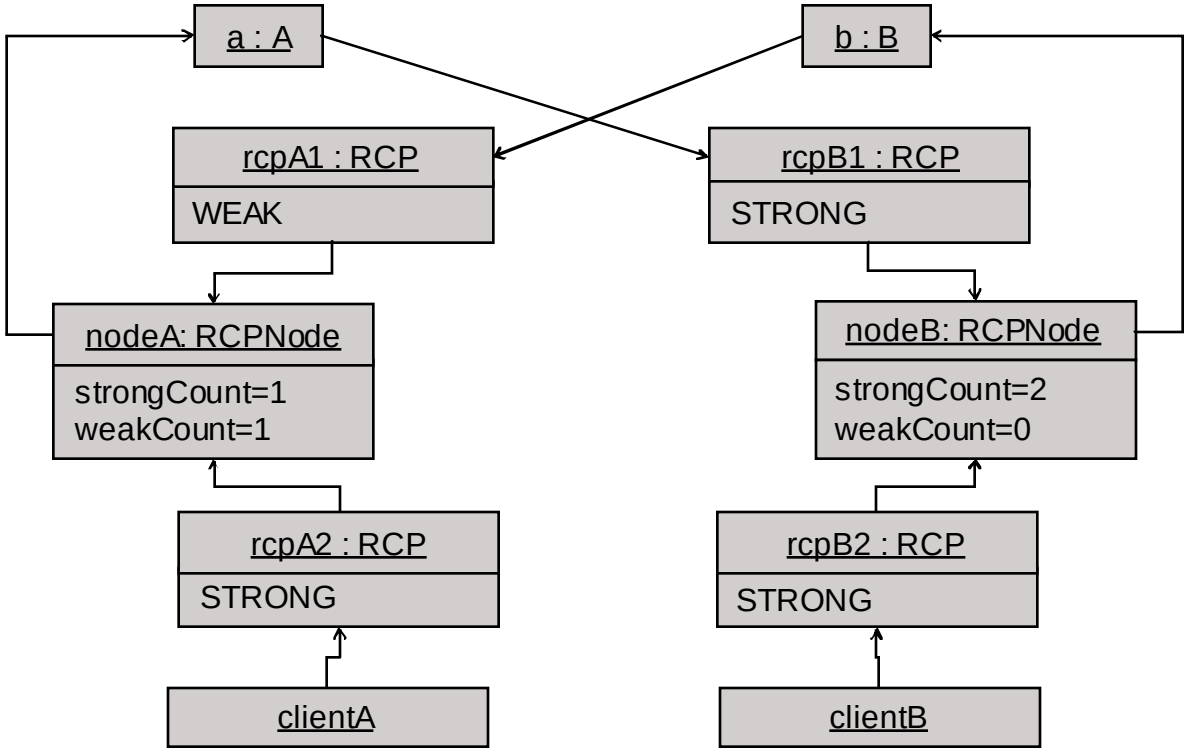
Source: teuchos/test/MemoryManagement/RCP_UnitTests.cpp

Run:

Build: teuchos/test/MemoryManagement/MemoryManagement_UnitTests.exe

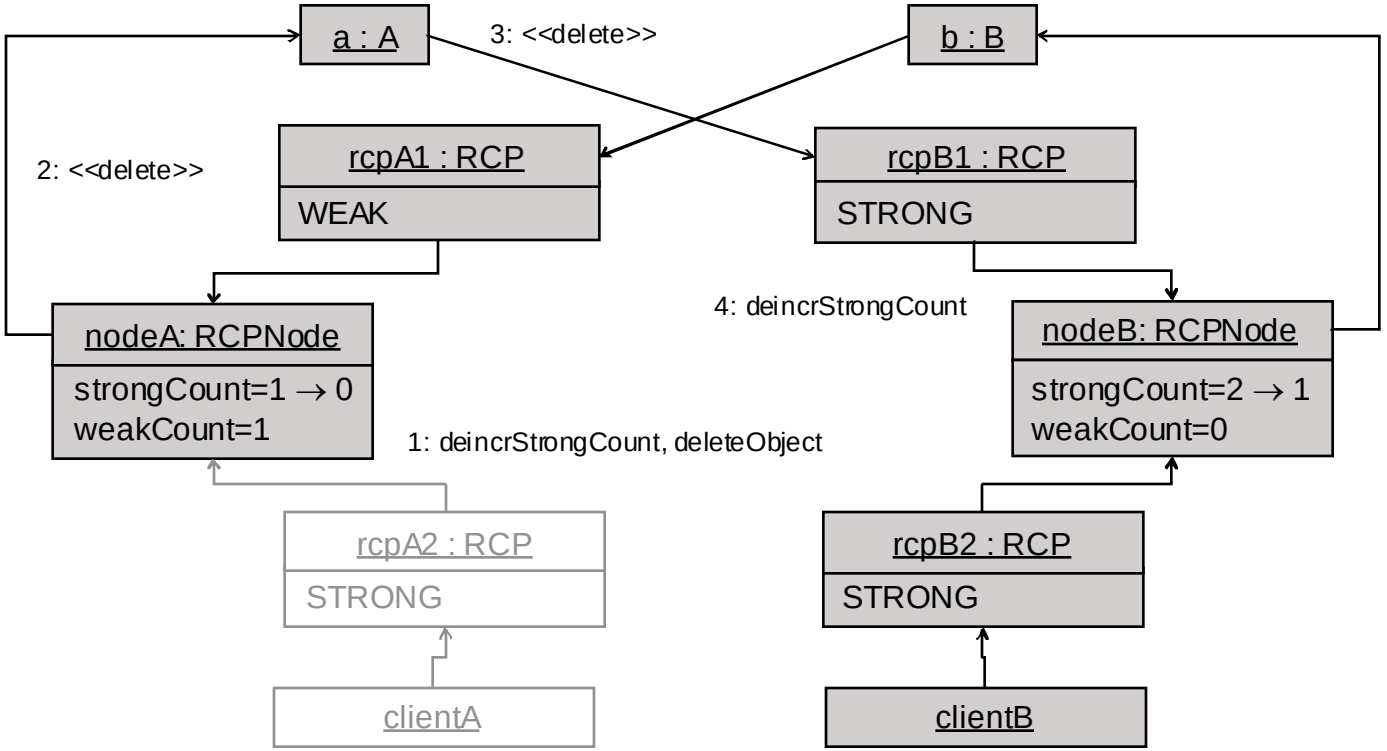
Sandia National Laboratories

Circular References with Weak/Strong RCPs



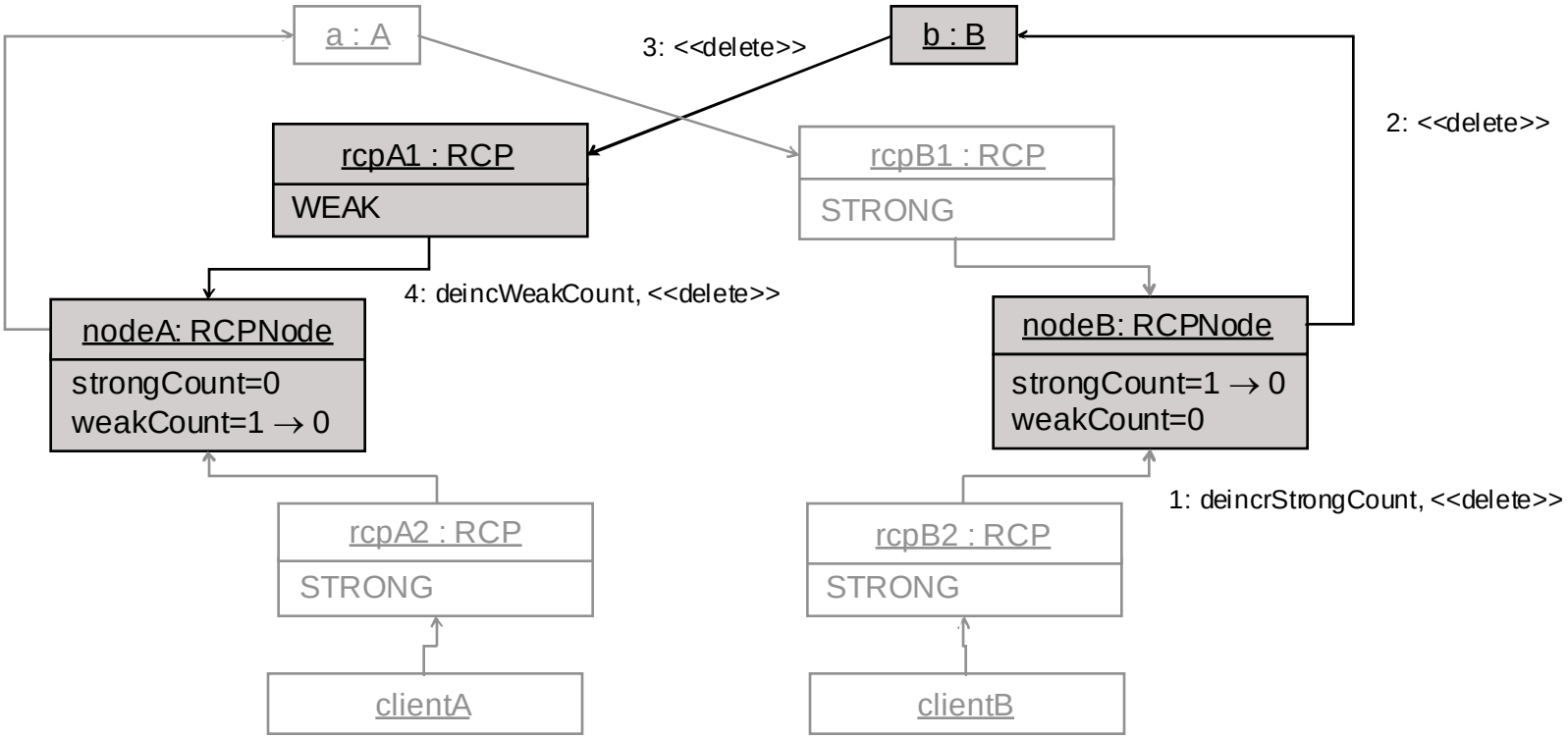
Sandia National Laboratories

Client A deleted first then Client B: #1



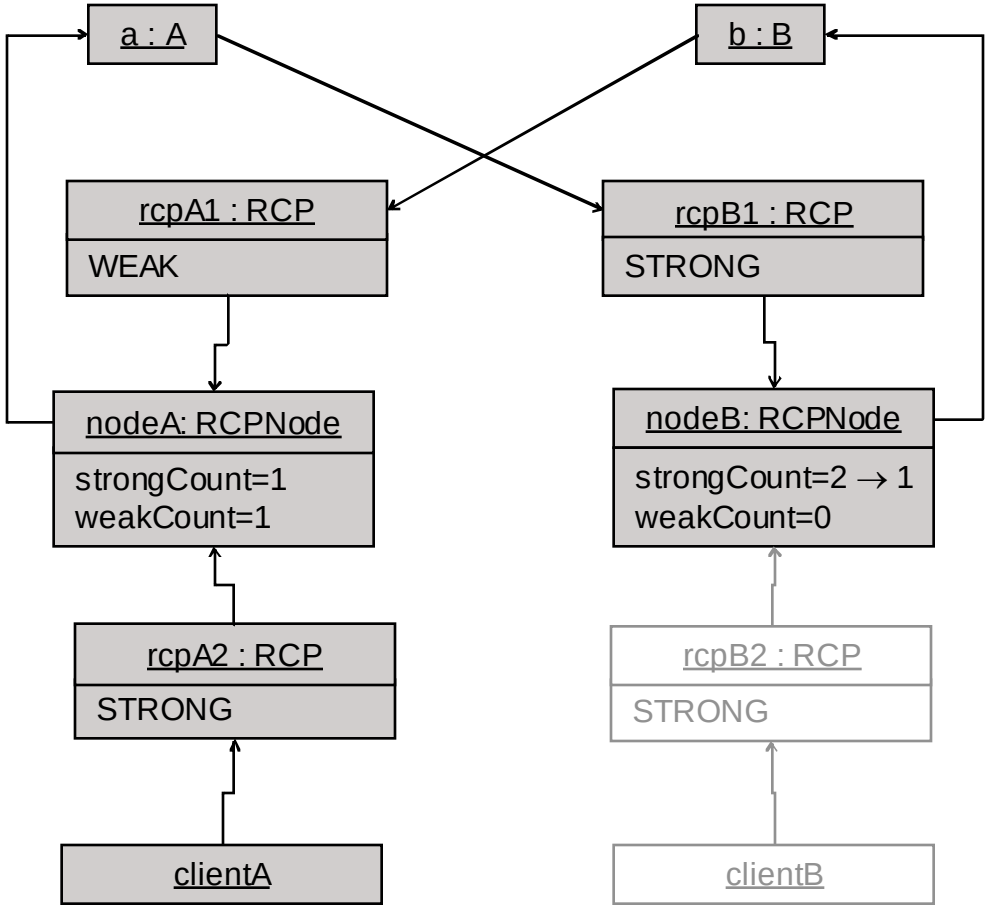
Sandia National Laboratories

Client A deleted first then Client B: #2



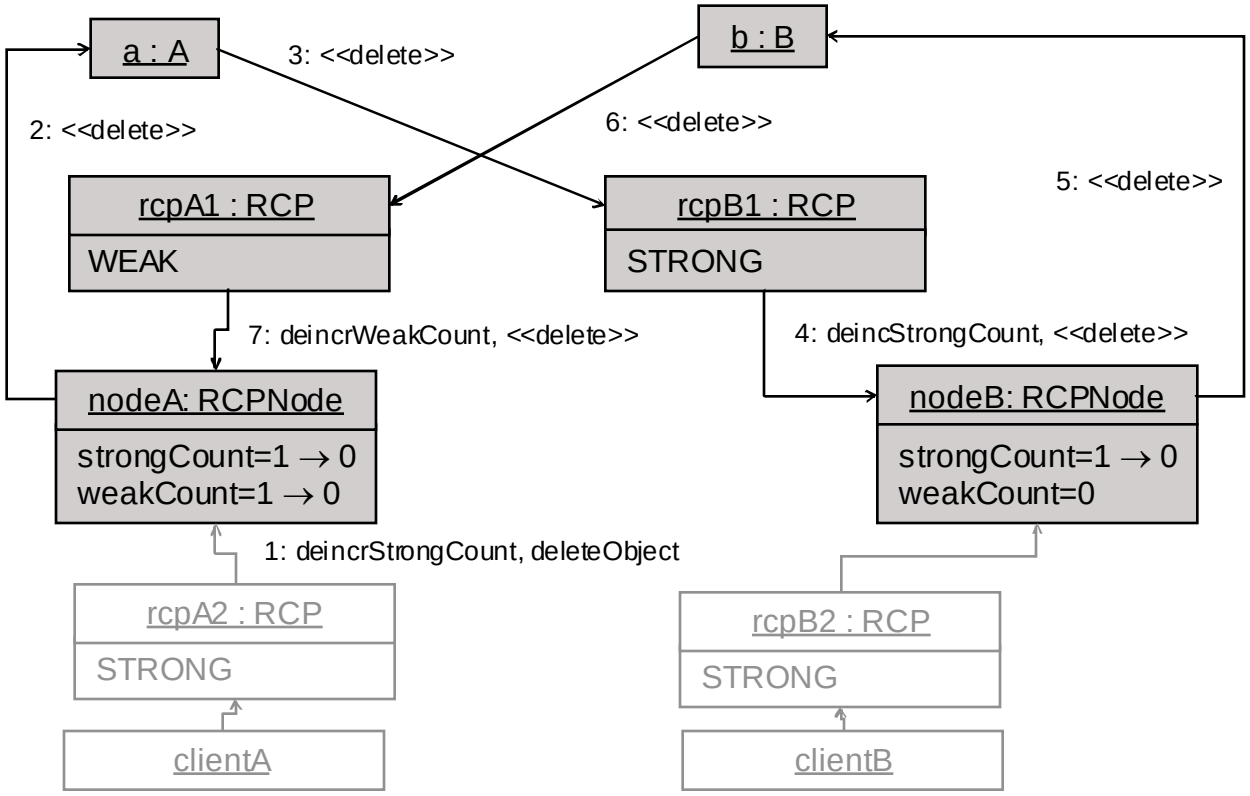
Sandia National Laboratories

Client B deleted first then Client A: #1



Sandia National Laboratories

Client B deleted first then Client A: #2



Sandia National Laboratories

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Runtime Checking of Dangling References: Unit Test Code

```
TEUCHOS_UNIT_TEST_TEMPLATE_1_DECL( ArrayRCP, danglingArrayView, T )
{
  ArrayView<T> av;
  {
    ArrayRCP<T> arcp1 = arcp<T>(n);
    av = arcp1();
  }
#ifdef TEUCHOS_DEBUG
  ...
  TEST_THROW( av[0], DanglingReferenceError );
 ...
#endif
}
```

See: Trilinos/packages/teuchos/test/MemoryManagement/ArrayRCP_UnitTests.cpp

Sandia National Laboratories

Runtime Checking of Dangling References: Error Output

```
Test that code {av[0];} throws Teuchos::DanglingReferenceError: passed

Exception message for expected exception:

 /cygdrive/c/_mystuff/PROJECTS/Trilinos.base/Trilinos/packages/teuchos/src/Teuchos_RCPNode.hpp:340:

 Throw number = 4

 Throw test that evaluated to true: true

 Error, an attempt has been made to dereference the underlying object
 from a weak smart pointer object where the underly object has already
 been deleted since the strong count has already gone to zero.

 Context inforamtion:

   RCP type:             Teuchos::ArrayView<double>
   RCP address:          0x22c5ac
   RCPNode type:         Teuchos::RCPNodeTmpl<double, Teuchos::DeallocArrayDelete<double> >
   RCPNode address       0xc33af8
   RCP ptr address:      0xc33aa0
   Concrete ptr address: 0xc33aa0

 Hint: Open your debugger and set conditional breakpoints in the various
 routines involved where this node object is first created with this
 concrete object and in all of the RCP objects of the type given above
 use this node object.  Debugging an error like this may take a little work
 setting up your debugging session but at least you don't have to try to
 track down a segfault that would occur otherwise!
```

Sandia National Laboratories

Outline

- Background
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Handling of circular references
  - Runtime checking of dangling references
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Class Data Member Conventions for Arrays

- Uniquely owned array, expandable (and contractible)

```
Array<T> a_;
```

- Shared array, expandable (and contractible)

```
RCP<Array<T> > a_;
```

- Shared array, fixed size

```
ArrayRCP<T> a_;
```

  - Advantages:
    - Your class object can allocate the array as `arcp(size)`
    - Or, you class object can accept a pre-allocated array from client

      => Allows for efficient views of larger arrays
    - The original array will be deleted when all references are removed!

Warning! Never use `Teuchos::ArrayView<T>` as a class data member!

  - `ArrayView` is never to be used for a persisting relationship!
  - Also, avoid using `ArrayView` for stack-based variables

Sandia National Laboratories

Function Argument Conventions : Single Objects, Value or Reference

- Non-changeable, non-persisting association, required

```
const T &a
```

- Non-changeable, non-persisting association, optional

```
const Ptr<const T> &a
```

- Non-changeable, persisting association, required or optional

```
const RCP<T> &a
```

- Changeable, non-persisting association, optional

```
const Ptr<T> &a
```

- Changeable, non-persisting association, required

```
const Ptr<T> &a
```

or

```
T &a
```

- Changeable, persisting association, required or optional

```
const RCP<const T> &a
```

Even if you don't want to use these conventions you still have to document these assumptions in some way!

Increases the vocabulary of you program! => Self Documenting Code!

Sandia National Laboratories

Function Argument Conventions : Arrays of Value Objects

- Non-changeable elements, non-persisting association

  `const ArrayView<const T> &a`

- Non-changeable elements, persisting association

  `const ArrayRCP<const T> &a`

- Changeable elements, non-persisting association

  `const ArrayView<T> &a`

- Changeable elements, persisting association

  `const ArrayRCP<T> &a`

- Changeable elements and container, non-persisting association

  `const Ptr<Array<T> > &a`

  or

  `Array<T> &a`

- Changeable elements and container, persisting association

  `const RCP<Array<T> > &a`

Warning!

- Never use `const Array<T>&` => use `ArrayView<const T>&`
- Never use `RCP<const Array<T> >&` => use `ArrayRCP<const T>&`

Function Argument Conventions : Arrays of Reference Objects

- Non-changeable objects, non-persisting association

  ```
  const ArrayView<const Ptr<const A> > &a
  ```

- Non-changeable objects, persisting association

  ```
  const ArrayView<const RCP<const A> > &a
  ```

- Non-changeable objects, changeable pointers, persisting association

  ```
  const ArrayView<RCP<const A> > &a
  ```

- Changeable objects, non-persisting association

  ```
  const ArrayView<const Ptr<A> > &a
  ```

- Changeable objects, persisting association

  ```
  const ArrayView<const RCP<A> > &a
  ```

- Changeable objects and container, non-persisting association

  ```
  Array<Ptr<A> > &a
  ```
  or
  ```
  const Ptr<Array<Ptr<A> > > &a
  ```

- Changeable objects and container, non-persisting container, persisting objects

  ```
  Array<RCP<A> > &a
  ```
  or
  ```
  const Ptr<Array<RCP<A> > > &a
  ```

- Changeable objects and container, persisting assoc. container and objects

  ```
  const RCP<Array<RCP<A> > > &a
  ```

- And there are other use cases!

Sandia National Laboratories

Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Challenges for Incorporating Teuchos Utility Classes

- More classes to remember
  - However, this increases the vocabulary of your programming environment!

    => More self documenting code!

- Implicit conversions not supported as well as for raw C++ pointers
  - Avoid overloaded functions involving these classes!

- Refactoring existing code?
  - First, we maintain backward compatibility!
  - Internal Trilinos code? => Not so hard but we need to be careful
  - External Trilinos (user) code? => Harder to upgrade “published” interfaces but manageable [Folwer, 1999]

Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
- Challenges to using Teuchos memory management utility classes
- Wrap up

Sandia National Laboratories

Teuchos classes verses boost/C++0x classes

- Teuchos provides complete system of low-level types to replace all raw C++ pointers in high-level C++ code
  - => Avoids all raw pointers at application level => safer code
  - => Boost and C++0x do not
- Teuchos classes throw exceptions in debug mode
  - => Makes unit tests easier to write
  - => Boost classes can be made to? Not sure about compatibility issues?
  - => Not sure of g++ checked STL can?
- Teuchos reference-counting classes have optional debug tracking mode to catch and diagnose circular references
  - => Helps to diagnose tricking circular reference problem (e.g. NOX, Tpetra, AztecOO/Thyra adapters)
  - => Nothing like this in boost (yet). => Might use sp_scalar_constructor_hook(...)?
- Teuchos reference-counted classes are two-way compatible with Boost/C++0x reference-counted classes
  - e.g. see teuchos/test/MemoryManagement/RCP_test.cpp
  - You don't have to pick on implementation of for all code!
- We control Teuchos, we can't control/change boost
  - => Modifying our own version of boost classes would be incompatible with other code
  - => Can't assume other code has not also used the "hooks"

- You can't mix and match Teuchos view classes and boost/C++0x classes and have strong debug runtime checking => Internal details must be shared!

Sandia National Laboratories

Next Steps

- Do detailed performance study on different platforms/compilers
- Write a detailed technical report describing these memory management classes
- Encourage the assimilation of these classes into more Trilinos and user software (much like was done for Teuchos::RCP)
  - Prioritize what to refactor based on risk and other factors

Make memory leaks and segfaults a rare occurrence!

Conclusions

- Using raw pointers at too high of a level is the source of nearly all memory management and usage issues in C++ (e.g. memory leaks and segfaults)
- Memory checking tools like Valgrind and Purify will never be able to sufficiently verify our C++ programs
- Boost and C++0x libraries do not provide a sufficient integrated solution
- `Teuchos::RCP` has been effective at reducing memory leaks of all kinds but we still have segfaults (e.g. array handling, off-by-one errors, etc.)
- New Teuchos classes `Array`, `ArrayRCP`, `ArrayView`, and `Tuple`, allow for safe (debug runtime checked) use of contiguous arrays of memory but very high performance in an optimized build
- Much Trilinos software will be updated to use these new classes
- Deprecated features will be maintained along with a process for supporting smooth and safe user upgrades
- A detailed technical report will be written to explain all of this

Sandia National Laboratories

THE END

References:

[Martin, 2003] Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003

[Meyers, 2005] Scott Meyers, *Effective C++: Third Edition*, Addison-Wesley, 2005

[Sutter & Alexandrescu, 2005], *C++ Coding Standards*, Addison-Wesley, 2005

[Fowler, 199] Martin Fowler, *Refactoring*, Addison-Wesley, 1999