SAND2007-7237C

# New Teuchos Utility Classes for Safer Memory Management in C++

**Roscoe A. Bartlett**
**Department of Optimization & Uncertainty Estimation**

**Sandia National Laboratories**

**Trilinos Users Group Meeting, November 6th, 2007**

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 …
- 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 additional Teuchos utility classes simular to Teuchos::RCP to encapsulate usage of raw C++ pointers for:
    - handling of single objects
    - handling of contiguous arrays of objects

# 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

# Outline

- Background
  - Background on C++
  - Problems with using raw C++ pointers at the application programming level
- 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

# Popularity of Programming Languages

| Position Oct 2007 | Position Oct 2006 | Delta in Position | Programming Language | Ratings Oct 2007 | Delta Oct 2006 | Status |
|---|---|---|---|---|---|---|
| 1 | 1 | = | Java | 21.616% | +0.44% | A |
| 2 | 2 | = | C | 14.591% | -3.07% | A |
| 3 | 5 | ↑↑ | (Visual) Basic | 11.166% | +1.44% | A |
| 4 | 3 | ↓ | C++ | 9.584% | -1.48% | A |
| 5 | 4 | ↓ | PHP | 9.498% | -0.36% | A |
| 6 | 6 | = | Perl | 5.351% | -0.12% | A |
| 7 | 8 | ↑ | C# | 3.740% | +0.68% | A |
| 8 | 7 | ↓ | Python | 3.433% | -0.03% | A |
| 9 | 9 | = | JavaScript | 2.685% | +0.48% | A |
| 10 | 13 | ↑↑↑ | Ruby | 2.386% | +1.30% | A |
| 11 | 12 | ↑ | PL/SQL | 1.966% | +0.87% | A |
| 12 | 15 | ↑↑↑ | D | 1.594% | +0.96% | A |
| 13 | 10 | ↓↓↓ | Delphi | 1.539% | -0.61% | A |
| 14 | 11 | ↓↓↓ | SAS | 1.383% | -0.67% | A |
| 15 | 14 | ↓ | ABAP | 0.849% | +0.20% | A- |
| 16 | 18 | ↑↑ | COBOL | 0.683% | +0.14% | B |
| 17 | 48 | ↑↑↑↑↑↑↑↑↑↑ | Lua | 0.596% | +0.53% | B |
| 18 | 16 | ↓↓ | Lisp/Scheme | 0.572% | -0.05% | B |
| 19 | 17 | ↓↓ | Ada | 0.559% | 0.00% | B |
| 20 | 21 | ↑ | Fortran | 0.446% | +0.05% | B |

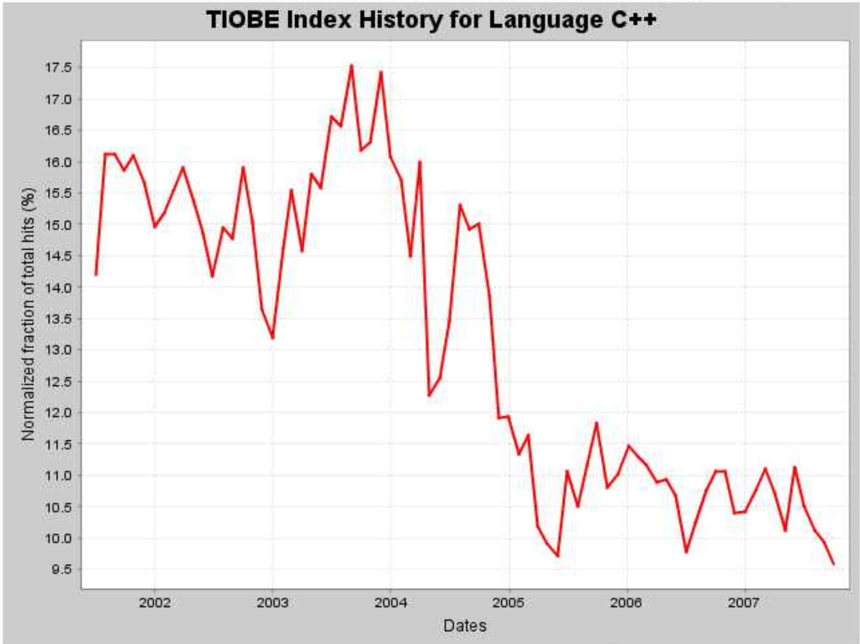
The ratings are based on:

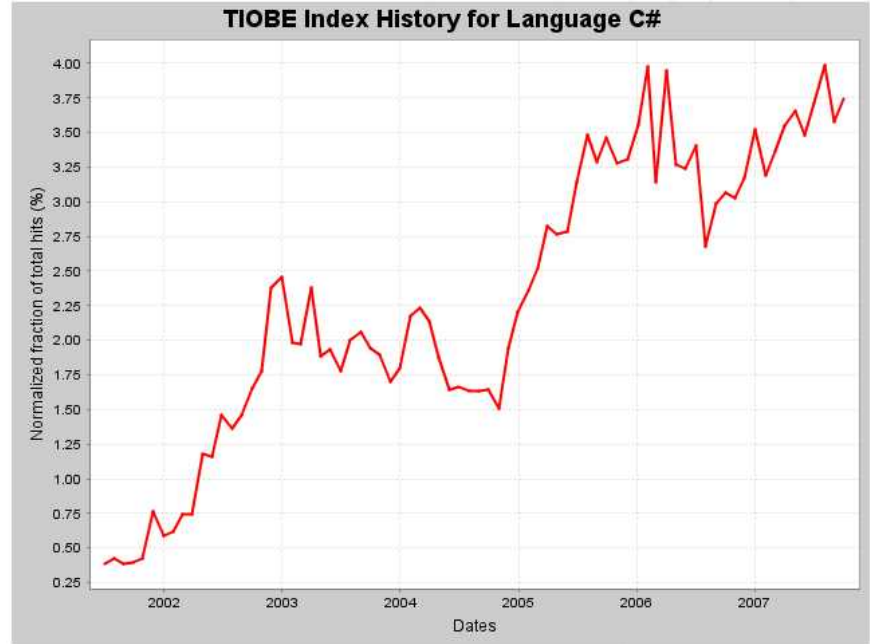
- world-wide availability of skilled engineers
- available courses
- third party vendors

- C++ is only the 4th most popular language
- C is almost twice as popular as C++ (so much for object-oriented programming)
- Java and Visual Basic popularity together are at least 4 times more popular than C++
- Fortran is hardly a blip
  - C++ is 20 times more popular
  - Java is 40 times more popular

Source: http://www.tiobe.com

# Declining Overall Popularity of C++




**The C++ Programming Language**

- Highest Rating (since 2001): 17.531% (3rd position, August 2003)
- Lowest Rating (since 2001): 9.584% (4th position, October 2007)

**The C# Programming Language**

- Highest Rating (since 2001): 3.987% (7th position, August 2007)
- Lowest Rating (since 2001): 0.384% (22nd position, August 2001)

- C++ is about half as popular as it was 4 years ago!
  - => Is C++ is on it's way out? => Of course not, but it's popularity is declining!
- C# is more than twice as popular as it was 4 years ago
  - => Will C# mostly replace C++? => Depends if C# expands past .NET!

Source: http://www.tiobe.com

Sandia National Laboratories

# Implications for the Decline in Popularity of C++

- Fewer and lower-quality tools for C++ in the future for:
  - Debugging?
  - Automated refactoring?
  - Memory usage error detection?
  - Others?
- Fewer new hirers will know C++ in the future
  - Bad news since C++ is already very hard to learn in the first place!
    - Who is going to take over the maintenance of our C++ codes?
  - However, the extremely low and declining popularity of Fortran does not stop organizations from using it either …

# The Good and the Bad for C++ for Scientific Computing

- The good:
  - Better ANSI/ISO C++ compilers now available for most of our important platforms
    - GCC is very popular for academics, produces fast code on Linux
    - Red Storm and the PGI C++ compiler
    - etc …
  - Easy interoperability with C, Fortran and other languages
  - Very fast native C++ programs
  - Precise control of memory (when, where, and how)
  - Support for generics (i.e. templates), operator overloading etc.
    - Example: Sacado! Try doing that in another language!
  - If Fortran is so unpopular then why are all of our customers using it?

    => C++ will stay around for a long time if we are productive using it!
- The bad:
  - Language is complex and hard to learn
  - Memory management is still difficult to get right

# Preserving our Productivity in C++ in Modern Times

- Support for modern software engineering methodologies
  - Test Driven Development (easy)
  - Other modern software engineering practices (code reviews supported by coding standards, etc.)
  - Refactoring => No automated refactoring tools!
- Safe memory management
  - Avoiding memory leaks
  - Avoiding segmentation faults from improper memory usage
- Training and Mentoring?
  - There is not silver bullet here!

# Refactoring Support: The Pure Nonmember Function Interface Idiom

**SANDIA REPORT**

SAND2007-4078
Unlimited Release
Printed October 2007

SAND2007-4078

**The Pure Nonmember Function Interface Idiom for C++ Classes**

Roscoe A. Bartlett

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.



Sandia National Laboratories

- Unifies the two idoms:
  - Non -Virtual Interface (NVI) idiom [Meyers, 2005], [Sutter & Alexandrescu, 2005]
  - Non-member Non-friend Function idiom [Meyers, 2005], [Sutter & Alexandrescu, 2005]
- Uses a uniform nonmember function interface for very “stable” classes (see [Martin, 2003] for this definition of “stable”)
- Allows for refactorings to non-public virtual functions without breaking current client code
- Doxygen \relates feature attaches link to nonmember functions to the classes they are used with.

# Outline

- Background
  - Background on C++
  - Problems with using raw C++ pointers at the application programming level
- 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

# 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!
    ```

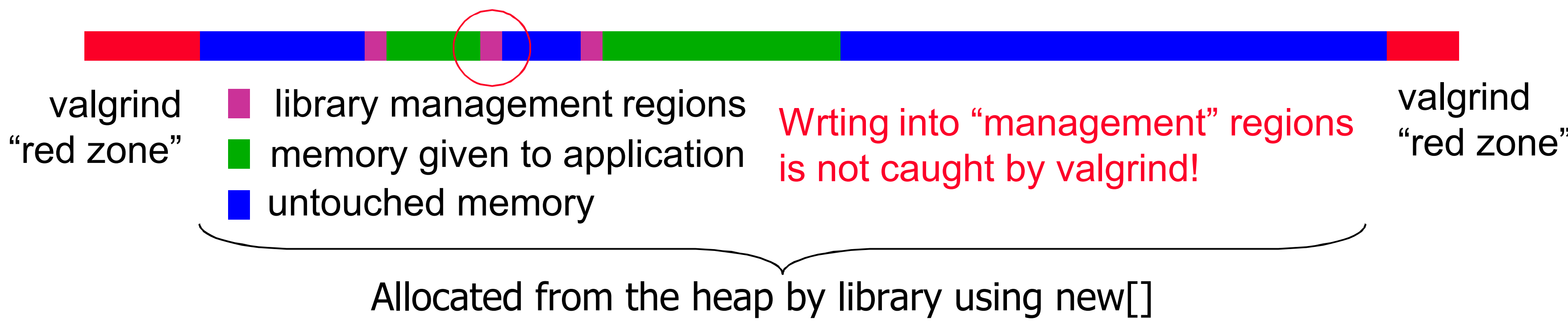
- How do we fix this?
  - Memory leaks? => Reference counting! (not a 100% guarantee)
  - Segfaults? => Memory checkers like Valgrind and Purify? (far from a 100% guarantee)

# Ineffectiveness of Memory Checking Utilities

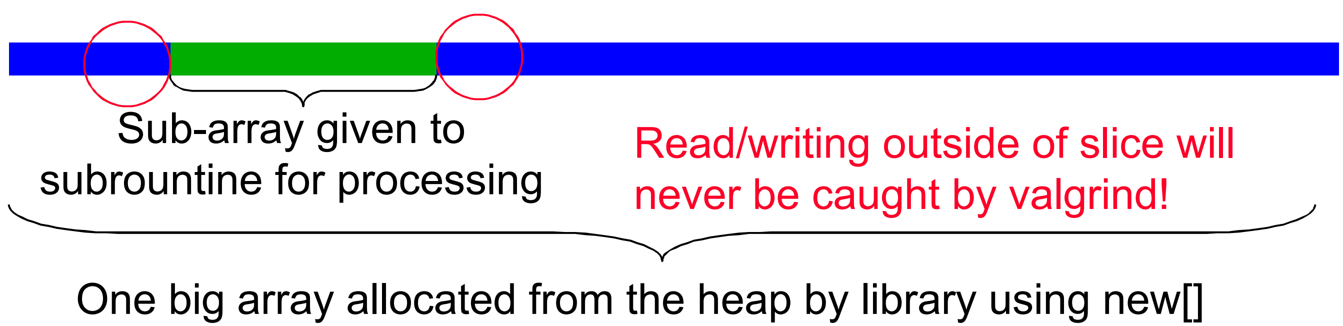
- Memory checkers like Valgrind and Purify only know about stack and heap memory requested from the system!

  => 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
- 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

# Memory Management: Safety vs. Cost, Flexibility, and Control

- How important is a 100% guarantee that memory will not be misused?
- Two kinds of features (i.e. guarantees)
  - Memory access checking (e.g. array bounds checking etc.)
  - Memory cleanup (e.g. garbage collection)
- Extreme approaches:
  - C: All memory is handled by the programmer, few if any language tools for safety
  - Python: All memory allocation and usage is controlled and/or checked by the runtime system
- With a 100% guarantee comes with a cost in:
  - Speed: Checking all memory access at runtime can be expensive (e.g. Matlab, Python, etc.)
  - Flexibility: Can't place objects where ever we want to (e.g. no placement new)
  - Control: Controlling exactly when memory is acquired and given back to the system (e.g. garbage collections running at bad times can kill parallel scalability)

# Memory Management Philosophy: The Transportation Analogy

- Little regard for safely, just speed: Riding a motorcycle on the interstate (no helmet, 100 MPH, doing a wheelie, in heavy traffic)

  => Coding in C/C++ with only raw pointers at the application programming level

- An almost 100% guarantee: Driving a reinforced tank (Styrofoam suite, racing helmet, Hans neck system, 10 MPH max speed)

  => All coding in a language like Java or Python

- Reasonable safety precautions (not 100%), and speed: Driving in a car (using a seat belt, driving speed limit, defensive driving, etc.)

  How do we get there? => We can get there from either extreme …

  – Sacrificing speed & efficiency for safely: Go from the motorcycle to the car:

    => Coding in C++ with memory safe utility classes

  – Sacrificing some safely for speed & efficiency: Going from the tank to the to the car:

    => Python or Java for high-level code, C/C++ for time critical operations

Before we make a mad rush to Java/Python for the sake of safer memory usage lets take another look at making C++ safer

# Outline

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

# Semantics of STL Containers: std::vector

`std::vector<T>` for continuous data

- Stored data type T must be a value type
  - Default constructor: `T::T()`
  - Copy constructor: `T::T(const T&)`
  - Assignment operator: `T& T::operator=(const T&)`
- Non-const std::vector<T>

  ```
  std::vector<T> v;
  ```

  - Can change shape of the container (add elements, remove elements etc.)
  - Can change element objects
- Const std::vector<T>

  ```
  const std::vector<T> &cv = v;
  ```

  - Can not change the shape of the container
  - Can not change the elements
  - Can only read elements (e.g. `val = cv[i]`);

# General Problems with using std::vector at Application Level

- Usage of std::vector is not checked

```
std::vector<T> v;
…
a[i]; // Unchecked
*(a.begin()+i); // Unchecked
for ( … ; a1.begin() != a2.end() ; … ) { … } // Unchecked
```

- What about std::vector::at(i)?

```
// Are you going to write code like this?
#ifdef DEBUG
  val = a.at(i); // Really bad error message if throws!
#else
  val = a[i];
#endif
```

- What about checking iterator access? => There is no equivalent to at(i)
- Specialized STL memory allocators disarm memory checking tools!
- What about a checked implementation of the STL?
  - Item 83, *C++ Coding Standards*: "Use a checked STL implementation"
  - A checked STL implementation is hard to come by, especially for GNU/Linux
  - This has to be part of your everyday programming toolbox!

Sandia National Laboratories

# Problems with using std::vector as Function Arguments


- Using a raw pointer to pass in an array of objects to modify

  void foo ( T v[], const int n )

  – Allows function to modify elements (good)
  – Allows for views of larger data (good)
  – Requires passing the dimension separately (bad)
  – No possibility for memory usage checking (bad)

- Using a std::vector to pass in an array of objects to modify

  void foo( std::vector<T> &v )

  – This allows functions to modify elements (good)
  – Keeps the dimension together with data (good)
  – Allows function to also add and remove elements (usually bad)
  – Requires copy of data for subviews (bad)

- Using a std::vector to pass in an array of const objects

  void foo( const std::vector<T> &v )

  – Requires copy of data for subviews (bad)
  – You are throwing away 95% of the functionality of std::vector!

Yes there is an std::valarray class but that has lots of problems too!

# Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

# 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)
  - Containers (arrays, maps, lists, etc.)
  - Views of arrays (persisting and non-persisting associations)
  - 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 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

# Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - 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
  – Used to be called Teuchos::RefCountPtr
    - See the script teuchos/refactoring/change-RefCountPtr-to-RCP-20070619.sh

- 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
  – Integrated with other memory utility classes

# 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

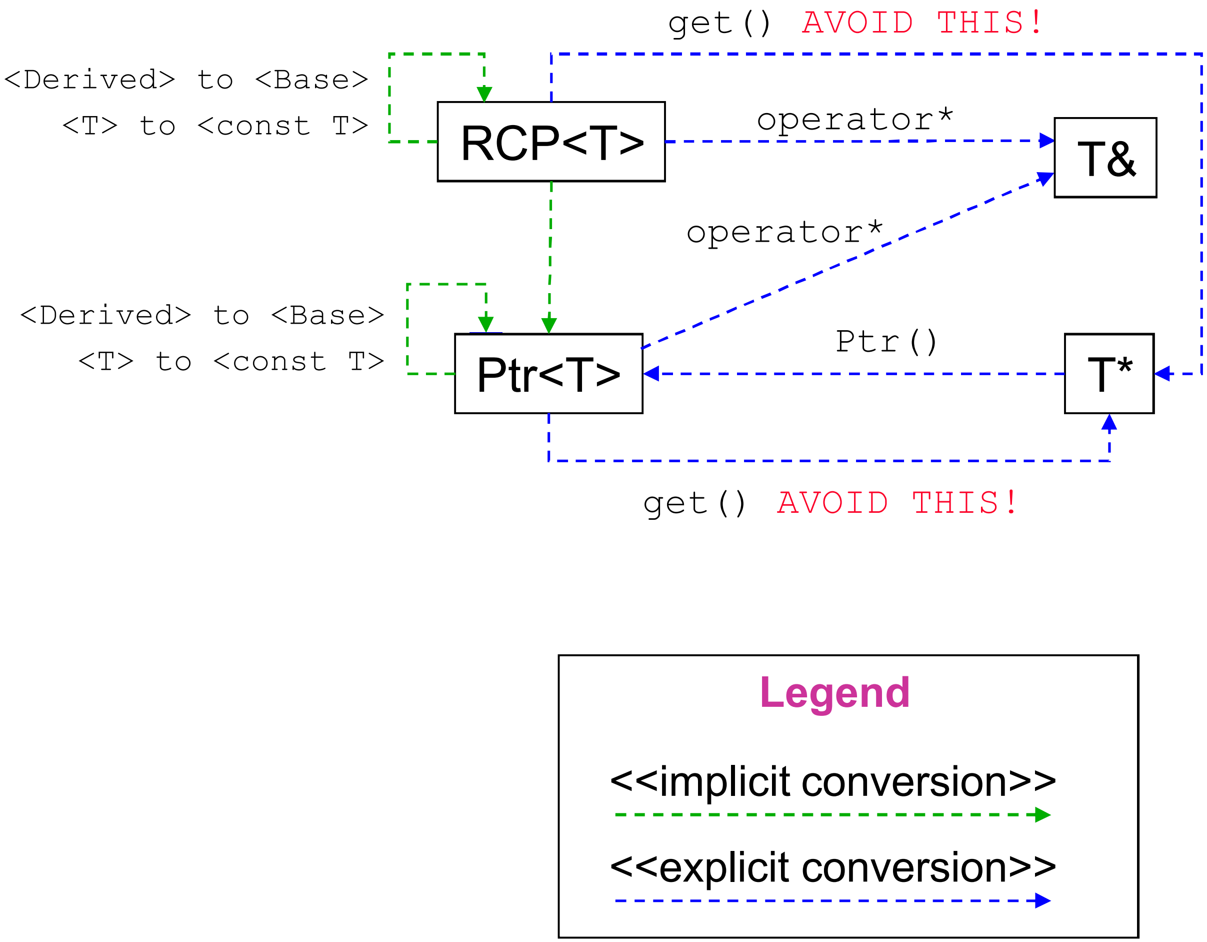
Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.



Sandia National Laboratories

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

# Conversions Between Single-Object Memory Management Types



# Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

# Utility Classes for Memory Management of Arrays of Objects

- Teuchos::ArrayView (New class)

```
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

- Teuchos::ArrayRCP (Failry new class)

```
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

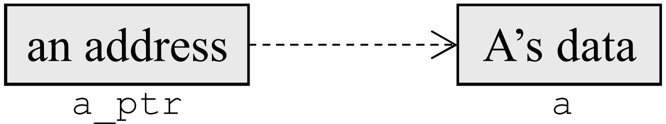
- Teuchos::Array (Existing class but majorly reworked)

```
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

# Raw Pointers and [Array]RCP : const and non-const

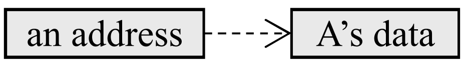
Example:
```
A  a;
A* a_ptr = &a;
```




**Important Point:** A pointer object `a_ptr` of type `A*` is an object just like any other object with value semantics and can be const or non-const

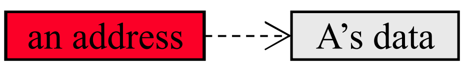
| Raw C++ Pointers | | RCP |
|---|---|---|
| `typedef A* ptr_A;` | equivalent to | `RCP<A>` |
| `typedef const A* ptr_const_A;` | equivalent to | `RCP<const A>` |

Remember this equivalence!



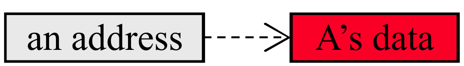

non-const pointer to non-const object

| | | |
|---|---|---|
| `ptr_A a_ptr;` | equivalent to | `RCP<A> a_ptr;` |
| `A * a_ptr;` | | |



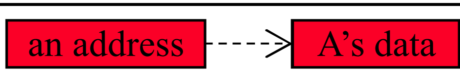

const pointer to non-const object

| | | |
|---|---|---|
| `const ptr_A a_ptr;` | equivalent to | `const RCP<A> a_ptr;` |
| `A * const a_ptr;` | | |




non-const pointer to const object

| | | |
|---|---|---|
| `ptr_const_A a_ptr;` | equivalent to | `RCP<const A> a_ptr;` |
| `const A * a_ptr;` | | |




const pointer to const object

| | | |
|---|---|---|
| `const ptr_const_A a_ptr;` | equivalent to | `const RCP<const A> a_ptr;` |
| `const A * const a_ptr;` | | |

# Teuchos::ArrayRCP

```
template<class T>
class ArrayRCP {
private:
   T *ptr_; // Non-debug implementation
  Ordinal lowerOffset_;
  Ordinal upperOffset_;
   RCP_node *node_; // Reference counting machinery
```

- General purpose replacement for raw C++ pointers to deal with contiguous arrays of data and uses reference counting
- Supports all of the good pointer operations for arrays and more:

```
++ptr, --ptr, ptr++, ptr--, ptr+=i // Increments to the pointer
*ptr, ptr[i] // Element access (debug checked)
ptr.begin(), ptr.end() // Returns iterators (debug checked)
```

- Support for const and non-const:

```
ArrayRCP<T>                // non-const pointer, non-const elements
const ArrayRCP<T>          // const pointer, const elements
ArrayRCP<const T>          // non-const pointer, const elements
const ArrayRCP<const T> // const pointer, const elements
```

- Does not support bad pointer array operations:

```
ArrayRCP<Base> p2 = ArrayRCP<Derived>(rawPtr); // No compile!
```

- ArrayRCP is reused for all checked iterator implementations!

Sandia National Laboratories

# Teuchos::ArrayView

```
template<class T>
class ArrayView {
private:
   T *ptr_; // Non-debug implementation
  Ordinal size_;
```

- Light-weight replacement for raw C++ pointers to deal with contiguous arrays of data for use as function arguments
- Only support array dereferencing and iterators:

```
ptr[i] // Dereferencing the pointer to access elements
ptr.begin(), ptr.end() // Returns iterators (debug checked)
```

- Uses ArrayRCP for checked implementation!
- Support for const and non-const element access

```
ArrayView<T>            // non-const elements
ArrayView<const T>      // const elements
```

# Teuchos::Array

```
template<class T>
class Array {
private:
   std::vector<T> vec_; // Non-debug implementation
```

- Thin, inline wrapper around std::vector
- Debug checked element access:

```
a[i] // Debug runtime checked
a[-1] // Throws exception in debug build!
a[a.size()] // Throws exception in debug build!
```

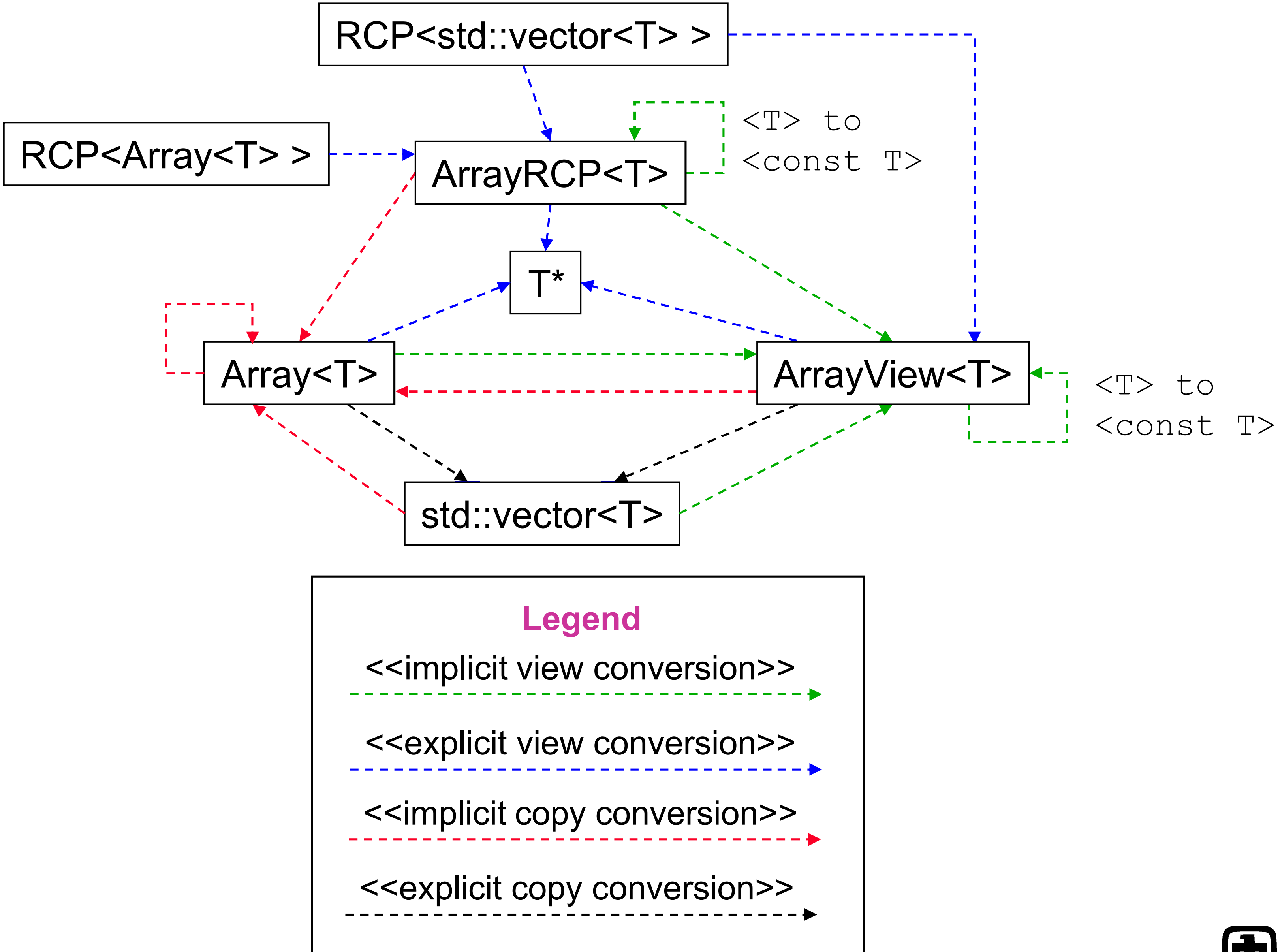
- Debug checked iterators (uses ArrayRCP):

```
*(ptr.begin()+i) // Debug runtime checked
*(ptr.begin-1) // Throws exception in debug build!
*(ptr.end()) // Throws exception in debug build!
```

- Conversions to and from std::vector
- Nonmember constructors

```
Array<T> a = tuple(obj1,obj2,…);
```

# Conversions Between Array Memory Management Types



# Outline

- Background
- High-level philosophy for memory management
- Existing STL classes
- Overview of Teuchos Memory Management Utility Classes
  - Introduction
  - Management of single objects
  - Management for arrays of objects
  - Usage of Teuchos utility classes as data objects and as function arguments
- Challenges to using Teuchos memory management utility classes
- Wrap up

# Class Data Member Conventions for Arrays

- Uniquely owned array, expandable (and contractable)

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

- Shared array, expandable (and contractable)

```
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 <u>never</u> to be used for a persisting relationship!

  – Also, avoid using `ArrayView` for stack-based variables

# Function Argument Conventions : Single Objects, Value or Reference

- Non-changeable, non-persisting, required

```
const T &a
```

- Non-changeable, non-persisting, optional

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

- Non-changeable, persisting , required or optional

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

- Changeable, non-persisting, optional

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

- Changeable, non-persisting, required

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

or

```
T &a
```

- Changeable, persisting, required or optional

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

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

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

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, persisting association

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

- Changeable elements and container, persisting associations

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

- And there are other use cases!

# 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

# 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?
  - 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]

How can we smooth the impact of these and other refactorings?

# Refactoring, Deprecated Functions, and User Support

- How can we refactor existing code?

  => Keep deprecated functions but ifdef them (supported for one release cycle?)

- Example: Existing Epetra function:

```
class Epetra_MultiVector {
public:
  ReplaceGlobalValues(int NumEntries, double *Values, int *Indices);
};
```

- Refactored function:

```
class Epetra_MultiVector {
public:
  // New function
  ReplaceGlobalValues(const ArrayView<const double> &Values,
    const ArrayView<const int> &Indices);
#ifdef TRILINOS_ENABLE_DEPRICATED_FEATURES
  // Depricated function
 ReplaceGlobalValues(int NumEntries, double *Values, int *Indices)
  { ReplaceGlobalValues(arrayView(Values,NumEntries),
      arrayView(Indices,NumEntries)); }
#endif
};
```

- How does this help users?

# Refactoring, Deprecated Functions, and User Support

Upgrade process for user code:

1. Add -DTRILINOS_ENABLE_DEPRICATED_FEATURES to build Trilinos and user code
2. Test user code (should compile right away)
3. Selectively turn off -DTRILINOS_ENABLE_DEPRICATED_FEATURES in user code and let compiler show code that needs to updated, Example:

```
// UserFunc.cpp
#undef TRILINOS_ENABLE_DEPRICATED_FEATURES
#include "Epetra_MultiVector.hpp"
void foo( Epetra_MultiVector &V )
{
  std::vector<double> values(n); …
  std::vector<double> indices(n); …
  V.ReplaceGlobalValues(n,&values[0],&indices[0]); // No compile
}
```

4. Fix a few function calls, Example:

```
  V.ReplaceGlobalValues(values,indices); // Now this will compile!
```

5. Turn -DTRILINOS_ENABLE_DEPRICATED_FEATURES back on and recompile
6. Run user tests and get all of them to pass before moving on [Fowler, 1999]
7. Repeat steps 3 through 6 for all user code until all deprecated calls are gone!

User code is incrementally and safely upgraded over time!

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

# Next Steps

- Finish development and testing of these Teuchos memory management utility classes (arrays of contiguous memory)
- Incorporate them into a lot of Trilinos software
  - Initially: teuchos, rtop, thyra, stratimikos, rythmos, moocho, …
  - Get practical experience in the use of the class and refine design
- 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 based on risk and other factors
- Start developing other memory safe utility classes:
  - Teuchos::Map: Safe wrapper around std::map
  - Teuchos::List: Safe wrapper around std::list
  - Others?

Make memory leaks and segfaults a rare occurrence!

# Conclusions

- Using raw C++ pointers at too high of a level is the source of nearly all memory management and usage issues (e.g. memory leaks and segfaults)
- STL classes are not safe and their use can make code actually less safe than when using raw C++ pointers (i.e. library handled memory allocation)
- Memory checking tools like Valgrind and Purify will never be able to sufficiently verify our C++ programs
- Declining popularity of C++ means we will have less support for tools for refactoring, debugging, memory checking, etc.
- 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, and ArrayView allow for safe (debug runtime checked) use of contiguous arrays of memory
- 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
- More memory-safe classes will be added in the future

# 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

Sandia National Laboratories

# Extra Slides

# Reasonable Precautions: C++ with Memory Safe Utility Classes vs. Python Mixed with C/C++

- Pure C++ program with memory safe classes
  - Advantages:
    - Native code gives instant performance
    - One standard compiler, less mixed-language issues
  - Disadvantages:
    - Top level code is not 100% safe
- Java/Python mixed with C/C++
  - Advantages
    - Top level code is nearly 100% safe
  - Disadvantages
    - Native code is slow
    - Mixed language, tools support problems, etc.

Before we make a mad rush to Java/Python for the sake of safer memory usage lets take another look at making C++ safer

# Value Semantics vs. Reference Semantics

## A. Value Semantics

```
class S {
public:
  S();                       // Default constructor
  S(const S&);               // Copy constructor
  S& operator=(const S&); // Assignment operator
  …
};
```

- Used for small, concrete datatypes
- Identity determined by the value in the object, not by its object address (e.g. obj==1.0)
- Storable in standard containers (e.g. std::vector<S>)
- Examples: int, bool, float, double, char, std::complex, extended precision …

## B. Reference Semantics

```
class A {
public:
  // Pure virtual functions
  virtual void f() = 0;
  …
};
```

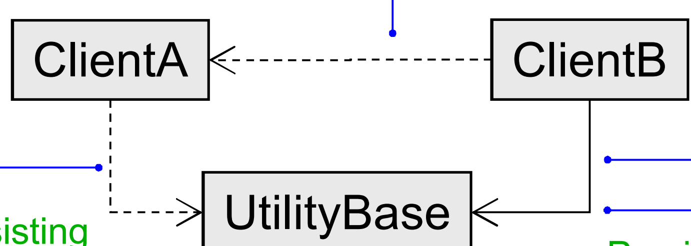
- Abstract C++ classes (i.e. has pure virtual functions) or for large objects
- Identity determined by the object's address (e.g. &obj1 == &obj2)
- Can not be default constructed, copied or assigned (not storable in standard containers)
- Examples: std::ostream, any abstract base class, …

# Persisting vs. Non-Persisting Associations

- Non-persisting association: An object association that only exists within a single function call and no "memory" of the object persists after the function exits
- Persisting association: An object association that exists beyond a single function call and where some "memory" of the object persists
- Examples:

```
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};

class ClientB {
  UtilityBase *utility_;
public:
  ClientB() : utility_(0) {}
  ~ClientB() { delete utility_; }
  void initialize( UtilityBase *utility ) { utility_ = utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};
```




Non-persisting association

UML class diagram

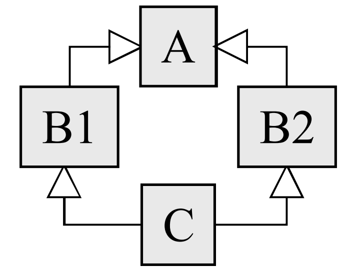
Persisting association

- Non-persisting associations:
  - Use C++ references and Teuchos::Ptr
- Persisting associations:
  - Use Teuchos::RCP

Sandia National Laboratories

# Teuchos::RCP

- RCP combines concepts of "smart pointers" and "reference counting" to build an imperfect but effective "garbage collection" mechanism in C++

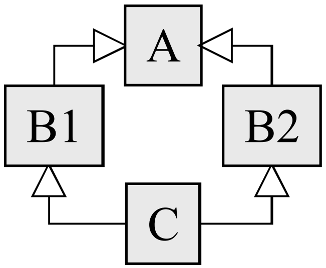



- Smart pointers mimic raw C++ pointer usage and syntax
  – Value semantics: i.e. default construct, copy construct, assignment etc.
  – Object dereference: i.e. (*ptr).f()
  – Pointer member access: i.e. ptr->f()
  – Conversions :
    - Implicit conversions using templated copy constructor: i.e. C* to A*, and A* to const A*
    - Explicit conversions: i.e. rcp_const_cast<T>(p), rcp_static_cast<T>(p), rcp_dynamic_cast<T>(p)
- Reference counting
  – Automatically deletes wrapped object when last reference (i.e. smart pointer) is deleted
  – Watch out for circular references! These create memory leaks!
  – Tip: Define the macro TEUCHOS_SHOW_ACTIVE_RCP_NODES
- RCP<T> is not a raw C++ pointer!
  – Implicit conversions from T* to RCP<T> and visa versa are not supported!
  – Failure of implicit casting and overload function resolution!
  – Other problems …
- Advanced Features
  – Template deallocation policy object
    - Allows other an delete to be called to clean up
    - Allows one smart pointer (i.e. boost::shared_ptr) to be embedded in a RCP
  – Extra data
    - Allows RCP to wrap classes that do not have good memory management (e.g. old Epetra)
    - Allows arbitrary events to be registered to occur before or after the wrapped object is deleted

# Implicit Casting with RCP : Common Problems/Mistakes

## Passing RCP by non-const reference instead of by const reference

Programming mistake!




```
void foo7(RCP<A> &a);
void foo7(const RCP<A> &a);

void boo4() {
  RCP<C> c = rcp(new C);
  RCP<A> a = c;
  foo7(a); // Okay, no cast
  foo7(c); // Error, can not cast involving non-const reference
  foo7(c); // Okay, implicit case involving const reference okay
}
```

## Failure to perform implicit conversion with overloaded functions

A deficiency of smart pointers over raw pointers

```
RCP<A>       foo9(const RCP<A>       &a);
RCP<const A> foo9(const RCP<const A> &a);

RCP<A> boo5() {
  RCP<C> c = rcp(new C);
  return foo9(c);          // Error, call is ambiguous!
  RCP<A> a = c;
  return foo9(a);          // Okay, calls first foo9(…)
  return foo9(rcp_implicit_cast<A>(c)); // Okay, calls first foo9(…)
}
```

Calls `foo9(A* a)` when using raw C++ pointers!

Sandia National Laboratories