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 7th, 2007
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 uninitializied 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 more 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
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>=</td>
<td>Java</td>
<td>21.616%</td>
<td>+0.44%</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>=</td>
<td>C</td>
<td>14.591%</td>
<td>-3.07%</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>↑↑</td>
<td>(Visual) Basic</td>
<td>11.166%</td>
<td>+1.44%</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>C++</td>
<td>9.584%</td>
<td>-1.48%</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>↓</td>
<td>PHP</td>
<td>9.499%</td>
<td>-0.96%</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>=</td>
<td>Perl</td>
<td>5.351%</td>
<td>-0.12%</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>↑</td>
<td>C#</td>
<td>3.760%</td>
<td>+0.68%</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>↓</td>
<td>Python</td>
<td>3.433%</td>
<td>-0.03%</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>=</td>
<td>JavaScript</td>
<td>2.665%</td>
<td>+0.48%</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>↑↑↑</td>
<td>Ruby</td>
<td>2.386%</td>
<td>+1.38%</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>↑</td>
<td>PL/SQL</td>
<td>1.966%</td>
<td>+0.87%</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>↑↑↑</td>
<td>D</td>
<td>1.594%</td>
<td>+0.96%</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>↓↓</td>
<td>Delphi</td>
<td>1.539%</td>
<td>-0.61%</td>
<td>A</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>↓↓</td>
<td>SAS</td>
<td>1.383%</td>
<td>-0.67%</td>
<td>A</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>↓</td>
<td>ABAP</td>
<td>0.849%</td>
<td>+0.20%</td>
<td>A-</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>↑↑</td>
<td>COBOL</td>
<td>0.683%</td>
<td>+0.14%</td>
<td>B</td>
</tr>
<tr>
<td>17</td>
<td>48</td>
<td>↑↑↑↑↑↑↑↑↑↑↑↑↑↑↓</td>
<td>Lua</td>
<td>0.595%</td>
<td>+0.53%</td>
<td>B</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>↓↓</td>
<td>Lisp/Scheme</td>
<td>0.572%</td>
<td>-0.05%</td>
<td>B</td>
</tr>
<tr>
<td>19</td>
<td>17</td>
<td>↓↓</td>
<td>Ada</td>
<td>0.579%</td>
<td>0.00%</td>
<td>B</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>↑</td>
<td>Fortran</td>
<td>0.445%</td>
<td>+0.05%</td>
<td>B</td>
</tr>
</tbody>
</table>

The ratings are based on:

- world-wide availability of skilled engineers
- available courses
- third party vendors
- only max of language dialects

- 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](http://www.tiobe.com)  
Referenced in appendix of [Booch, 2007]
Declining Overall Popularity of C++

The C++ Programming Language
- Highest Rating (since 2001): 17.531% (3rd position, August 2003)

- 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!

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 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](http://www.tiobe.com)
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

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

Approved for public release; further dissemination unlimited.

SAND2007-4078

- 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 virtual functions without breaking 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
    
    ```cpp
    int *ptr;
    ptr[20] = 5; // BANG!
    ```
  
  – Using to handle memory of single objects
    
    ```cpp
    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:
    
    ```cpp
    int *ptr = new int[n];
    // These are totally unchecked:
    *(ptr++), *(ptr--), ptr[i]
    ```
  
  – Creates memory leaks when exceptions are thrown:
    
    ```cpp
    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)

  ![Diagram showing memory regions managed by library]

  Allocated from the heap by library using new[]

  - Program managed memory

  ![Diagram showing memory regions managed by program]

  One big array allocated from the heap by library using new[]

  Memory checkers can never sufficiently verify your program!
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?
  – I will leave that as an open question for now

• 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 Metaphor

• Little regard for safely, just speed: Riding a motorcycle with no helmet, in heavy traffic, going 100 MPH, doing a wheelie

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

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

  => All coding in a fully checked language like Java, Python, or Matlab

• Reasonable safety precautions (not 100%), and good speed: Driving a car, wearing 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
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>
  
  ```cpp
  std::vector<T> v;
  ```
  – Can change shape of the container (add elements, remove elements etc.)
  – Can change element objects

• Const std::vector<T>
  
  ```cpp
  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[1];`);
General Problems with using std::vector at Application Level

- Usage of std::vector is not checked

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

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

```cpp
// 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?
  - “Use a checked STL implementation”: Item 83, C++ Coding Standards
  - A checked STL implementation is hard to come by, especially for GNU/Linux
  - This has to be part of your everyday programming toolbox!
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
Utility Classes for Memory Management of Single Classes

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

Approved for public release; further dissemination cleared.

Sandia National Laboratories

http://trilinos.sandia.gov/documentation.html
Conversions Between Single-Object Memory Management Types

Legend

<<implicit conversion>>

<<explicit conversion>>
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)
  
  ```cpp
def 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 (Fairly new class)

  ```cpp
  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)

  ```cpp
  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:

```c
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

<table>
<thead>
<tr>
<th>Non-const Pointer</th>
<th>Const Pointer</th>
<th>Const Pointer to Const Object</th>
<th>RCP Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typedef A* ptr_A;</code></td>
<td><code>typedef const A* ptr_const_A;</code></td>
<td></td>
<td><code>equivalent to RCP&lt;A&gt;</code></td>
</tr>
</tbody>
</table>

**Remember this equivalence!**

---

<table>
<thead>
<tr>
<th>Non-const Pointer</th>
<th>Const Pointer</th>
<th>Const Pointer to Const Object</th>
<th>RCP Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptr_A</code></td>
<td><code>const ptr_A</code></td>
<td><code>const A*</code></td>
<td><code>equivalent to const RCP&lt;A&gt;</code></td>
</tr>
<tr>
<td><code>A*</code></td>
<td><code>const A*</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Non-const Pointer</th>
<th>Const Pointer</th>
<th>Const Pointer to Const Object</th>
<th>RCP Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptr_const_A</code></td>
<td><code>const ptr_const_A</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td><code>const A*</code></td>
<td><code>const A*</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Non-const Pointer</th>
<th>Const Pointer</th>
<th>Const Pointer to Const Object</th>
<th>RCP Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const ptr_const_A</code></td>
<td><code>const A*</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td><code>const A * const</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
</tbody>
</table>
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!
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
template<class T>
class Array {
private:
    std::vector<T> vec_; // Non-debug implementation

    // Thin, inline wrapper around std::vector
    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

Legend

<<implicit view conversion>>
<<explicit view conversion>>
<<implicit copy conversion>>
<<explicit copy conversion>>
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 `rcp(size)`
    - Or, your 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
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
  ```
  or
  ```
  T &a
  ```
- Changeable, persisting association, required or optional
  ```
  const RCP<const T> &a
  ```

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

Even if you don’t want to use these conventions you still have to document these assumptions in some way!
Function Argument Conventions: Arrays of Value Objects

- Non-changeable elements, non-persisting association
  
  \[
  \text{const ArrayView<\text{const T}> \&a}
  \]

- Non-changeable elements, persisting association
  
  \[
  \text{const ArrayRCP<\text{const T}> \&a}
  \]

- Changeable elements, non-persisting association
  
  \[
  \text{const ArrayView<T> \&a}
  \]

- Changeable elements, persisting association
  
  \[
  \text{const ArrayRCP<T> \&a}
  \]

- Changeable elements and container, non-persisting association
  
  \[
  \text{const Ptr<Array\langle T\rangle > \&a}
  \]

  or
  
  \[
  \text{Array\langle T\rangle \&a}
  \]

- Changeable elements and container, persisting association
  
  \[
  \text{const RCP<Array\langle T\rangle > \&a}
  \]

**Warning!**

- Never use \text{const Array\langle T\rangle \&} \quad \Rightarrow \quad \text{use ArrayView<\text{const T}>\&}

- Never use \text{RCP<\text{const Array\langle T\rangle >}\&} \quad \Rightarrow \quad \text{use ArrayRCP<\text{const T}>\&}
Function Argument Conventions: Arrays of Reference Objects

- Non-changeable objects, non-persisting association
  
  \texttt{const ArrayView<const Ptr<const A> > \&a}

- Non-changeable objects, persisting association
  
  \texttt{const ArrayView<const RCP<const A> > \&a}

- Non-changeable objects, changeable pointers, persisting association
  
  \texttt{const ArrayView<RCP<const A> > \&a}

- Changeable objects, non-persisting association
  
  \texttt{const ArrayView<const Ptr<A> > \&a}

- Changeable objects, persisting association
  
  \texttt{const ArrayView<const RCP<A> > \&a}

- Changeable objects and container, non-persisting association
  
  \texttt{Array<Ptr<A> > \&a or const Ptr<Array<Ptr<A> > > \&a}

- Changeable objects and container, persisting association
  
  \texttt{Array<RCP<A> > \&a or const Ptr<Array<RCP<A> > > \&a}

- Changeable elements and container, persisting associations
  
  \texttt{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 and smooth the transition for dependent code?
  => Keep deprecated functions but ifdef them (supported for one release cycle?)

• Example: Existing Epetra function:

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

• Refactored function:

```cpp
class Epetra_MultiVector {
public:
    // New function
    ReplaceGlobalValues(const ArrayView<const double> &Values,
                         const ArrayView<const int> &Indices);
    #ifdef TRILINOS_ENABLE_DEPRECATED_FEATURES
    // Deprecated 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_DEPRECATED_FEATURES` to build Trilinos and user code
2. Test user code (should compile right away)
3. Selectively turn off `-DTRILINOS_ENABLE_DEPRECATED_FEATURES` in user code and let compiler show code that needs to updated, Example:

```cpp
// userFunc.cpp
#undef TRILINOS_ENABLE_DEPRECATED_FEATURES
#include "Epetra_MultiVector.hpp"
void userFunc( 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_DEPRECATED_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!**
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 classes and refine their 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 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
- More memory-safe classes will be added in the future
THE END

References:


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