**SANDIA REPORT**

2010-2234
Unlimited Release
Printed Feburary 2011

# Teuchos C++ Memory Management Classes, Idioms, and Related Topics

# The Complete Reference

## Trilinos version 11.0

### A Comprehensive Strategy for Safe and Efficient Memory Management in C++ for High Performance Computing

Roscoe 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.

Approved for public release; further dissemination unlimited.



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

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

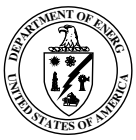
Printed in the United States of America. This report has been reproduced directly from the best available copy.

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

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

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

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



2010-2234
Unlimited Release
Printed Feburary 2011

# Teuchos C++ Memory Management Classes, Idioms, and Related Topics

# The Complete Reference

## Trilinos version 11.0

## A Comprehensive Strategy for Safe and Efficient Memory Management in C++ for High Performance Computing

Roscoe Bartlett

**Abstract**

The ubiquitous use of raw pointers in higher-level code is the primary cause of all memory usage problems and memory leaks in C++ programs. This paper describes what might be considered a radical approach to the problem which is to encapsulate the use of all raw pointers and all raw calls to `new` and `delete` in higher-level C++ code. Instead, a set of cooperating template classes developed in the Trilinos package Teuchos are used to encapsulate every use of raw C++ pointers in every use case where it appears in high-level code. Included in the set of memory management classes is the typical reference-counted smart pointer class similar to `boost::shared_ptr` (and therefore C++0x `std::shared_ptr`). However, what is missing in boost and the new standard library are non-reference counted classes for remaining use cases where raw C++ pointers would need to be used. These classes have a debug build mode where nearly all programmer errors are caught and gracefully reported at runtime, i.e. essentially eliminating undefined behavior related to memory usage. The default optimized build mode strips all runtime checks and allows the code to perform as efficiently as raw C++ pointers with reasonable usage. Also included is a novel approach for dealing with the circular references problem that imparts little extra overhead and is almost completely invisible to most of the code (unlike the boost and therefore C++0x approach). Rather than being a radical approach, encapsulating all raw C++ pointers is simply the logical progression of a trend in the C++ development and standards community that started with `std::auto_ptr` and is continued (but not finished) with `std::shared_ptr` in C++0x. The goal is nothing short of eliminating undefiend behavior related to memory usage from C++ programs without destroying the low-level control and performance of C++ programs.

# Contents


Preface ..... viii
1 Introduction ..... 1
2 Fundamental problems with raw C++ pointers ..... 5
2.1 Problems using raw C++ pointers for handling single objects ..... 5
2.2 Problems using raw C++ pointers for handling arrays of objects ..... 6
2.3 Problems with the incompatibility of `new/delete` and `try/throw/catch` ..... 7
3 Problems with common approaches for addressing memory management in C++ ..... 10
3.1 Problems with using `std::vector` for handling all arrays ..... 10
3.2 Problems with relying on standard memory checking utilities ..... 13
4 Important prerequisites ..... 16
4.1 Value types versus reference types ..... 16
4.2 Non-persisting versus persisting and semi-persisting associations ..... 18
5 Teuchos classes for safer memory management and usage ..... 23
5.1 Overview of basic approach employed by Teuchos memory management classes ..... 23
5.2 The proper role of raw C++ pointers ..... 26
5.3 Common aspects of all Teuchos memory management classes ..... 27
5.4 Memory management classes replacing raw pointers for single objects ..... 27
5.4.1 `Teuchos::Ptr<T>` ..... 28
5.4.2 `Teuchos::RCP<T>` ..... 30
5.4.3 Raw C++ references ..... 32
5.5 Memory management classes replacing raw pointers for arrays of objects ..... 32
5.5.1 `Teuchos::ArrayView<T>` ..... 34
5.5.2 `Teuchos::ArrayRCP<T>` ..... 36
5.5.3 `Teuchos::Array<T>` ..... 39
5.5.4 `Teuchos::Tuple<T,N>` ..... 41
5.5.5 Array views ..... 43
5.6 Const versus non-const pointers and objects ..... 44
5.7 Conversions ..... 45
5.7.1 Implicit and explicit raw-pointer-like conversions ..... 45
5.7.2 Conversions between different memory management types ..... 49
5.7.3 Implicit type conversion problems and shortcomings ..... 52
5.8 Core idioms for the use of the Teuchos memory management classes ..... 58
5.8.1 The non-member constructor function idiom ..... 58
5.8.2 General idioms for handling arrays of objects ..... 60
5.8.3 Idioms for class object data members and local variables ..... 61
5.8.4 Idioms for the specification of formal arguments for C++ functions ..... 63
5.8.5 Idioms for returning objects from C++ functions ..... 68
5.9 Reference-counting machinery in-depth ..... 76
5.9.1 Basic reference counting machinery ..... 76
5.9.2 Circular references and weak pointers ..... 79
5.9.3 Customized deallocators ..... 86
5.9.4 Embedded objects ..... 90
5.9.5 Extra data ..... 92
5.10 Roles and responsibilities for persisting associations: factories and clients ..... 94
5.11 Debug-mode runtime checking ..... 95
5.11.1 Detection of null dereferences and range checking ..... 95

5.11.2 Detection of circular references . . . 96
5.11.3 Detection of dangling references . . . 98
5.11.4 Detection of multiple owning RCP objects . . . 100
5.11.5 Performance of debug-mode checking versus memory checking tools . . . 102
5.11.6 Limitations of debug-mode runtime checking . . . 103
5.11.7 Exception handling and debugging . . . 105
5.12 Optimized performance . . . 107
5.12.1 Reference counting overhead . . . 108
5.12.2 Array access and iterator overhead . . . 113
5.12.3 Performance tuning strategies, semi-persisting associations . . . 114
5.13 Related idioms and design patterns . . . 119
5.13.1 The inverted object ownership idiom . . . 119
5.13.2 The separate construction and just-in-time initialization idioms . . . 121
5.13.3 The object self-reference idiom . . . 124
5.13.4 The generalized view design pattern . . . 128
5.14 Comparison with other class libraries and the standard C++ library . . . 139
5.15 Advice on refactoring existing software . . . 141
6 Miscellaneous topics . . . 144
6.1 Essential and accidental complexity, making implicit concepts explicit . . . 144
6.2 Philosophy of memory management: Safety, speed, flexibility and 100% guarantees . . . 146
7 Conclusions . . . 148
References . . . 149

**Appendix**

A Summary of Teuchos memory management classes and idioms . . . 151
B Commandments for the use of the Teuchos memory management classes . . . 159
C Argument for using a signed integer for size_type in the Teuchos array classes . . . 162
D Raw performance data . . . 166
D.1 Raw RCP performance data . . . 166
D.2 Raw Array performance data . . . 169

## Figures

1 UML Class Diagram showing non-persisting and persisting associations . . . 20
2 Conversions between different single-object memory management types. . . . 46
3 Conversions between array memory management types. . . . 47
4 Basic design of the Teuchos reference-counting machinery. . . . 76
5 Example of several RCP objects pointing to the same RCPNodeImpl object. . . . 78
6 Simple circular reference between two objects. . . . 80
7 Simple circular reference between two objects broken using a WEAK RCP. . . . 81
8 Weak pointer scenario where ClientA is deleted first . . . 82
9 Weak pointer scenario where ClientB is deleted first . . . 84
10 Example of a circular chain involving many objects and many classes. . . . 85
11 Example of duplicate owning RCPNodeImpl objects . . . 101
12 Timings for allocating and deallocating objects using RCP . . . 110
13 Timings of basic RCP operations on for three compilers . . . 112

14 Timings for basic Array, ArrayRCP, and ArrayView operations . . . . . 115
15 Depiction of contiguous and non-contiguous multi-vector column views. . . . . 129
16 Parent and child classes for “generalized view” design pattern. . . . . 130
17 State behavior for parent object in “generalized view” design pattern. . . . . 130

## Tables

1 Basic Teuchos memory management utility classes for encapsulating raw pointers. . . . . 24
2 Summary of capabilities of the basic Teuchos memory management classes. . . . . 24
3 Teuchos array container classes. . . . . 25
4 Common members and non-members for all types . . . . . 27
5 Additional non-members for array types . . . . . 33
6 Equivalences between raw pointer and smart pointer types for const protection . . . . . 45
7 Basic implicit and explicit conversions by smart-pointer types. . . . . 46
8 Summary of basic conversions supported involving single objects. . . . . 50
9 Summary of basic conversions supported for contiguous arrays. . . . . 51
10 Idioms for class data member declarations for value-type objects. . . . . 62
11 Idioms for class data member declarations for reference-types objects. . . . . 62
12 Idioms for passing value-type objects to C++ functions. . . . . 64
13 Idioms for passing reference-type objects to C++ functions. . . . . 65
14 Idioms for returning value-type objects from C++ functions. . . . . 69
15 Idioms for returning reference-type objects from C++ functions. . . . . 70
16 Overhead of runtime checking for serial Tpetra test suite. . . . . 103
17 Performance testing platforms. . . . . 107
18 Sizes of RCP and boost::shared_ptr objects for 64 bit GCC 4.1.2. . . . . 109

# Preface

This document describes the basic problems with raw memory management in C++ and presents an approach to addressing the problems by encapsulating all raw C++ pointers in a system of cooperating types. Almost every aspect is presented and issues of detailed usage, safety, performance, suggested idioms and many other topics are discussed. This is a fairly lengthy document with more than 140 pages in the main body and with 20 pages of appendices. This may seem like a lot of material to read through but consider that the 1500+ pages of mainstream literature on modern C++ usage in just the references [29, 31, 12, 26] still leaves a language that makes it too easy to write programs with undefined behavior (i.e. segfault) and memory leaks.

While this is a long document, there is a much shorter path given below that gives the basics for the anxious reader that does not need all the background material or the information on the many interesting side topic discussed.

**Abbreviated table of contents:**


- Section 1 "Introduction"
- Section 4.2 "Non-persisting versus persisting and semi-persisting associations"
- Section 5.1 "Overview of basic approach employed by Teuchos memory management classes"
- Section 5.4 "Memory management classes replacing raw pointers for single objects":
- Section 5.5 "Memory management classes replacing raw pointers for arrays of objects"
- Section 5.8 "Core idioms for the use of the Teuchos memory management classes"
- Section 5.10 "Roles and responsibilities for persisting associations: factories and clients"
- Section 5.14 "Comparison with other class libraries and the standard C++ library"
- Appendix A "Summary of Teuchos memory management classes and idioms"
- Appendix B "Commandments for the use of the Teuchos memory management classes"


The material shown above should be enough to a) give a basic idea of the motivation for the Techos memory management classes, b) describe the basic foundations for the classes, c) present the names, identities, and basic usage for each of the classes, d) describe the core idioms for the use of the classes, and e) mention how these classes compare with other classes in the Boost and the standard C++ libraries. This is the most basic material that should answer the most basic questions that most developers will have. The material in Appendix A and Appendix B should be used as a (relatively) short reference guide for the use of these classes. This material together with some existing code examples (e.g in Trilinos) should give an experienced C++ developer enough to get started using these classes in a productive way. Finally, the reader should go through the full table of contents at least once to get an idea of the variety of topics covered and where to look when more information is needed (and most developers will need to know most of this extra information at some point).

The rest of the material covered in this document either provides more background that might be needed to persuade some readers or expands on a number of topics that almost every developer will need to consider

at some point while using these classes including a) how to deal with type conversion problems, b) how the basic reference-counting machinery works, what types of debug-mode runtime checking is performed and how to debug problems when exceptions are thrown, c) what optimized performance should look like and how to better optimize code, d) what related idioms are useful or needed to fully exploit these classes, e) guidance on how to refactor existing software, and f) other related topics like a discussion of essential and accidental complexity.

Hopefully this document will be educational and help open the one's mind to what is possible to achieve in terms of safety and performance in modern C++ programs.

The classes and idioms described in this document have been encorporated into the coding guidelines document described in [5].

# 1 Introduction

A critical problem in computational science and engineering (CS&E) software as well as in other types of software developed in C++ is in the effective and safe management of memory and data. CS&E software often has the goal of high performance where arbitrary data copy leads to undue overhead and can actually complicate the software in many cases. It is common for CS&E software to share and pass around large blocks of memory in order to do work efficiently (however, common approaches such as described in [13] lead to many of the problems that exist in CS&E programs). At the most basic level, large arrays of integral and floating point data are managed along with more complex general objects and arrays of objects. In C++, the only universally accepted way to deal with memory for single objects and arrays of objects is to use raw C++ pointers. However, raw C++ pointer facilities for the manipulation and sharing of basic memory are inherently unsafe and error prone. The problem is further exacerbated when larger programs composed out of separately developed and maintained components are integrated together. Assumptions about the origin, ownership, and process for reclaiming memory and other resources remain the most basic problems with lower-level C++ programming techniques and are unfortunately still ubiquitous in the C++ community and even in the current C++ best-practices literature [31, 26]. The general C++ and CS&E communitie's inability to effectively address the basic problem of the usage of memory in large-scale modular C++ codes affects every aspect of software quality, productivity, reusabliltiy, and undermines the most basic software verification foundation for these codes. The challenges of writing software that uses raw memory management results in components that are overly rigid in how they can be used and reused which fundamentally detracts from the impact that such software could otherwise have and makes it more difficult develop and maintain. The problems created by the use of raw memory management can single-handily derail the vision of a large interconnected network of reusable CS&E software components developed and used by many different CS&E organizations [33]. Therefore, the issue of memory management has as much or more impact on the macroscopic properties of CS&E software components as it does on low-level internal software development.

C++ is a large and complex language that very few people really know how to use in a confident and successful way. Arguably the most serious problems in C++ are related to dynamic memory management (which must be used with any moderately complex object-oriented program) and raw access to memory (e.g. array and pointer usage errors). The built-in C++ support for dealing with dynamic memory allocation with `new` and `delete` is fundamentally incompatible with the built-in exception handling mechanism using `try`, `throw`, and `catch`. One cannot effectively build large-scale integrated software using just these low-level language features at the application programming level. Software developed this way yields undefined behavior (e.g. segfaults) and leaks memory unpredictably and is extremely difficult to integrate with other code. The only successful way to use C++ to create complex robust software is to develop and rigorously adopt a set of programming idioms for safely dealing with memory. By developing the right support software and associated idioms, we make C++ programs safer, better defined, faster to develop, and more efficient when run. The development of `std::vector` and `std::shared_ptr` are steps in the right direction by the C++ standards have not gone far enough (as will be explained below).

The reason that C++ is in this state of affairs is due to how C++ came into being and how it evolved over many years [28, 30]. C++ was first developed in the early 1980s as an extension to the popular C programming language and was first called "C with Classes". At the time, high efficiency, very low runtime overhead, and strong compatibility with C were critical requirements to the success of the new language. Without this, the original creator of C++, Bjarne Stroustrup, concluded that C++ would be "stillborn" [29]. The first C++ compilers were little more than preprocessors putting out C code on the back-end which was then compiled into executable binary code.

As the years went on, however, object-oriented programming was refined, computers become faster with more memory, and it was realized that more runtime support was required to enable more advanced usages of C++. As new features were added to C++ to support new programming idioms, a strong need for backward compatibility constrained the design of the language, sometimes making different language features incompatible when used together in raw form. The most unfortunate example of this, which was already mentioned, is the fundamental incompatibility of built-in dynamic memory management (i.e. using `new`/`delete`) and built-in exception handling (i.e. using `try`/`throw`/`catch`) that was added more than a decade later [28].

Because of the way that C++ "evolved" along with a strong need for backward compatibility, we have a language that is a disaster when used in raw form in undisciplined ways on complex large-scale software. Many programming teams have exploited this natural capability of C++ to create travesties of software which in turn have dumbfounded many a C++ programmer (and entire teams) and have resulted in giving C++ a bad name in the general software engineering community (see Section 6 "DDD Matters Today" in [1] and "The Case of the Construction Blob" in [15, Chapter 9] for a few examples of C++ bashing).

More specifically, using low-level manual memory management (e.g. `new`/`delete`, raw pointers everywhere) at all levels in C++ has resulted in several negative consequences in the development of C++ (and C) software that other more modern languages (e.g. Java and Python) have avoided:

- Programs that use raw memory management are more difficult to write and debug because it is difficult to track down invalid memory usage that results in undefined behavior (e.g. segfaults), double deletes, and memory leaks. (Also, memory checking tools like Valgrind and Purify are too slow and do not catch enough of these types of errors to adequately mitigate the problem.)
- Programs that use raw memory management can have many "hidden" memory usage errors (i.e. undefined behavior) that can linger in the code for months or years which damage the most basic foundations of software quality and verification. Many of these programs are "ticking time bombs" just ready to go off, sometimes with disastrous consequences for users and developers alike.
- Dealing with raw memory management at all levels consumes large amounts of developer focus which detracts from more general design focus. This results in software with lower quality designs compared to software written in other modern languages developed using the same amount of effort.
- Developers maintaining C++ programs that use raw memory management typically have a high degree of paranoia and fear about modifying the software (and for good reason because modifying such software is dangerous and error prone). This results in the tendency to not refactor software as requirements and domain knowledge change [14] which therefore results in software that dies the slow painful death of software entropy [8].
- Software that uses raw memory management at all levels necessarily have designs that overly constrain how the software is used and reused. For example, in such programs factory objects typically have to outlive the products they create and must also be responsible for deleting the objects they create. This results in large numbers of "static" factory objects that make the software hard to maintain and reuse in reasonable contexts.

The consequences of raw memory management described above are all too common in C++ development organizations and software produced by such organizations. This is why the general software development

community is largely moving away from C++ and instead moving to use more modern languages that do not require manual unchecked memory management [1].

However, C++ has some unique features that differentiate it from every other language in wide use which include:

- Strong typing (leads to high-performance code)
- High-performance native code
- Support for creating very efficient concrete data types with efficiency on par with built-in data types that do not require dynamic memory management (i.e. true stack-based objects)
- Support for operator overloading
- Support for object-oriented programming
- Support for generic programming (i.e. templates)
- A powerful turing-complete compile-time programming mechanism (i.e. template meta-programming)

No other programming language with wide availability has this powerful set of features. For instance, C++ can be used to create class libraries for capabilities like automatic differentiation [18] for computing derivatives of functions that achieves a level of generality and efficiency that has no rival in a software library in any other programming language (e.g., see the Trilinos[1] package Sacado[2] [27]). It is precisely the above feature set along with wide availability of high quality compilers on every major platform (including the cutting-edge massively parallel computers), good interoperability with other languages (through C interoperability), and strong support for next generation architectures [20] that makes C++ so attractive for writing computational science & engineering software in the first place.

It is also this unique feature set that is C++'s saving grace with respect memory management problems. In C++, one can actually develop a set of new data types that in essence can be used to develop new programming environments in C++. This essentially allows one to define a new programming language within C++ with a level of efficiency and flexibility that does not exist in any other programming language. This is exactly what this paper advocates with respect to basic memory management in C++; developing a new higher-level programming language in C++ for abstracting and encapsulating all raw memory usage as well as dynamic memory management that is very compatible with the built-in C++ exception handling mechanism. The approach being described here is really just the systematic and (arguably) elegant application to the approaches advocated in "Code Complete Second Edition" [24, Section 13.2: Pointers] for instance.

This paper describes a set of low-level C++ classes and supporting software in the Trilinos package Teuchos[3] that are used to encapsulate all raw pointers and enable strong debug-mode runtime checking while allowing for very high performance in non-debug-mode optimized builds. This ideas and approaches are built on the the ideas and foundations of standard C++ classes like `std::vector` and `std::shared_ptr` but fill in all of the gaps.

[1] `http://trilinos.sandia.gov`
[2] `http://trilinos.sandia.gov/packages/sacado/`
[3] `http://trilinos.sandia.gov/packages/teuchos/`

The Teuchos memory management classes and the idioms that they help to define (which are described in this paper) do not remove the need for programmers to learn and understand the intricate details of the C++ memory model and type system. On the contrary, learning to effectively use these memory management classes requires more effort over just learning raw C++. However, the payoff is that the programs that result from the use of these classes and idioms will be more likely to be correct on first writing, will be easier to debug when there are defects, will be easier and safer to maintain, and will be more self documenting (which helps all of the above). In fact, the self-documenting expressiveness of the resulting programs written using these classes and idioms is unmatched in any other programming language currently in popular use, including Java and Python. This statement will be backed up throughout this paper and then reiterated in Section 6.1.

The remainder of this paper assumes that the reader has some basic knowledge of C++ and is somewhat familiar with smart reference-counted pointer classes like `boost::shared_ptr` (which is the basis for the new C++0x `std::shared_ptr` class). The Teuchos equivalent for these smart pointer classes is `Teuchos::RCP` which is abbreviated here as just `RCP` in sample code. If the reader is not familiar with the basics of smart reference-counted pointer classes, then they should refer to [2] and [31]. If the reader is not familiar with fundamental C++ concepts like implicit type conversions, templates, object lifetime models, raw references and pointers and other basic topics, then some more basic background will be needed. However, specific references to basic C++ material in books like [26, 29, 31] are made throughout this document. So if the reader is a novice C++ programmer and is willing to look up the mentioned references, then this paper can be a good guide to help learn this basic C++ material as well.

A final warning: the material in this document is fairly detailed and will take a significant investment in time and experience writing code involving the Teuchos memory management classes using the idioms described here before a developer will be proficient. It takes years just to master raw C++ so it should be no surprise that learning a new set of idioms to fix a large number of the problems with raw C++ will also take a significant amount of time and effort. What is needed is a culture change in the C++ programming community where this type of approach and the idioms described here are taught at a very early stage; much like the STL is now being taught in introductory C++ courses. What we need is a revolution in C++ education but we have to start somewhere and that is what this paper is all about, getting started and on the road to a better generation of C++ programmers and C++ software. However, note that this document is not a tutorial but instead is a complete reference guide to the Teuchos memory management guide that covers almost every possible issue and reasonably related topic.

The body of this document is organized as follows. The fundamental problems with raw C++ pointers is described in Section-2. Common (suboptimal) approaches for addressing memory management problems are discussed in Section 3. Some important prerequisite concepts like value-types versus references-types and persisting versus non-persisting associations are defined in Section 4. With all this background and context in place, the Teuchos memory management classes are presented in Section 5. The basic outline of the approach in Section 5.1 is perhaps the first section one would jump to in order to get a quick idea what the Teuchos memory management classes are all about. Finally, taking a step back, the concepts of essential and accidental complexity and a philosophical discussion of the trade-offs between speed, safety and generality related to memory management are discussed in Section 6. Concluding remarks are given in Section 7.

# 2 Fundamental problems with raw C++ pointers

This section summarizes some of the fundamental problems with basic C++ features related to raw pointers. What is going to be argued is that while many people will claim that C++ pointers are strongly typed, it will be shown that raw pointers are actually very weakly typed in many respects and how this weak typing is the cause of many programming errors that result in incorrect programs and with undefined behavior (e.g. segfaults).

In the following examples, the simple classes shown in Listing 1 are used in demonstration code:

**Listing 1** *:*

```
class A {
  char dummy_;
  char *char_ptr_;
public:
  A(...);
  void incrementA() { ++(*char_ptr_); }
};

class B : public A {
  int size_;
  int *int_ptr_;
public:
  B(...);
  void incrementB() { ++(*int_ptr_); }
};
```

The concrete class hierarchy in Listing 1 was chosen to demonstrate some insidious and perhaps less well known flaws in the C++ type system when dealing with raw C++ pointers.

## 2.1 Problems using raw C++ pointers for handling single objects

There are a number of problems with using raw C++ pointers to manage single objects. For example, given a class object of type `B` in Listing 1 consider a pointer declared as:

```
B some_b(...);
B *b_ptr = &some_b;
```

Some of the legitimate things that one can do with this pointer are:

```
// Call member functions
b_ptr->incrementA();
b_ptr->incrementB();
// Extract reference
B &b_ref = *b_ptr;
```

```
// Copy pointer
B *b_ptr2 = b_ptr;
// Implicit conversion to const
const B *b_ptr3 = b_ptr;
// Implicit conversion to base type
A *a_ptr4 = b_ptr;
```

However, nothing good can ever come of any of the following operations when a pointer is only pointing to a single object:

```
b_ptr++
b_ptr--
++b_ptr
--b_ptr
b_ptr+i
b_ptr-i
b_ptr[i]
```

No C++ compiler I have ever worked with will even issue a warning when array operations are invoked on a raw C++ pointer for which it is clear is only pointing to a single object.

The problem here of course is that there is no way to tell the C++ compiler that a raw pointer is only pointing to a single object. With respect to differentiating single objects and arrays of objects, C++ pointers are untyped and the compiler provides no help whatsoever in statically asserting correct usage. This is strike one for the notion that C++ pointers are strongly typed!

## 2.2 Problems using raw C++ pointers for handling arrays of objects

When considering the semantics of raw C++ pointers one realizes that raw pointers are really designed primarily for dealing with contiguous arrays of objects (save for one exception that is mentioned below). This is because almost every operation that C++ defines for raw pointers makes sense and is fairly well defined when raw C++ pointers are pointing with contiguous arrays of objects. Every valid C++ operation will not be reviewed for raw pointers to contiguous arrays of objects (see [29] for a complete listing). Instead, a few examples are shown where the C++ type system using raw pointers falls flat on its face when dealing with arrays of memory.

One particularly troubling example where the C++ type system fails when dealing with raw C++ pointers to contiguous arrays of memory is shown in Listing 2.

**Listing 2** *:*

```
void foo(const int n) {
    B *b_array = new B[n];
    A *a_array = b_array; // Compiles just fine :-(
    for (int i = 0; i < n; ++i) {
      a_array[i].incrementA(); // KABOMMMMM!
```

```
    }
    delete [] b_array;
}
```

There are a lot of beginning and even some more experienced C++ programmers that would think that the C++ code in Listing 2 is just fine. The resulting program has undefined behavior and may seem to run okay in some cases but in the above case will almost certainly segfault right away. The above code fragment is wrong, wrong, wrong as described in [12, Gotcha 33] and [31, Item 100]. Without going into great detail, converting from a pointer for an array of type `B` to a pointer of type of base type `A` is almost always asking for disaster because the alignment of the base type `A` will be wrong according to the full type `B` (again see [12, Gotcha 33] all the gory details). As a result, for the second iteration `i=1`, the embedded pointer in `a_array[1].char_ptr_` is pointing to garbage because on most 32 bit machines with most compilers, the address in `a_array[1].char_ptr_` is actually the binary representation of the integer `b_array[0].size_`. Therefore, calling `a_array[1]->incrementA()` on most 32 bit machines is equivalent to performing:

```
++(*reinterpret_cast<char*>(b_array[0].size_)); // KABOMMMMM!
```

If this sort of thing comes as a surprise to C++ developers, then they should probably fear using raw memory in C++ more than they currently do and should seriously consider using the safer approach to encapsulating raw memory usage that is being advocated in this paper.

So how did C++ come to allow such completely wrong and dangerous operations like shown in Listing 2? It is because of the untyped dual nature of raw C++ pointers in trying to handle both single objects and contiguous arrays of objects with one data type where the full set of operations are not appropriate for either. The ability to cast raw C++ pointers from derived types to base types only ever generally makes sense when the pointer is pointing to a single object and will not be interpreted as a pointer to a contiguous array of objects. Note that C does not have this problem since there is no such thing as type derivation and the designers of C never even envisioned that raw C pointers would be used for such a thing. However, when the original designer of C++ adopted the C type system along with raw pointers and tried to apply it to an object-oriented language, he inadvertently opened up a number of serious language gotchas that we are still living with to this day. This is another strike for the notion that C++ pointers are strongly typed!

### 2.3 Problems with the incompatibility of `new/delete` and `try/throw/catch`

The use of raw pointers and raw calls to `new` and `delete` is also fundamentally incompatible with the built-in C++ exception handling mechanism using `try`/`throw`/`catch`. For example, the following code will leak memory if the function `someFunc()` throws a C++ exception:

```
void foo()
{
  A *a = new A(...);
  someFunc(); // Could throw an exception
  delete a; // Will never be called if someFunc() throws!
}
```

According to current C++ best practices relating to memory management and exception handling as described in [26, Item 29] and [31, Item 71], code like shown above that leaks memory is totally unacceptable in production quality C++ programs. This fundamental incompatibility of the built-in C++ dynamic memory management facilities using `new`/`delete` and the built-in exception handling mechanism using `try`/`throw`/`catch` was clear even to the committee that created the official 1998 C++ standard. However, again, because of the need for backward compatibility they were powerless to fix the problem at the language level. Instead, the C++ standards committee included the first standard C++ smart pointer class; `auto_ptr`. The class `auto_ptr` solves only the most basic problem with raw C++ pointers and that is that it ensures that memory will be reclaimed when exceptions are thrown. For example, the following refactored function will not leak any memory when `someFunc()` throws:

```
void foo()
{
  std::auto_ptr<A> a(new A(...));
  someFunc(); // Could throw an exception
  // NOTE: delete will get called on the A object no matter how this
  // function exists (i.e. normal exit or with a throw) since it is
  // called by the destructor of the stack object 'a' of type
  // std::auto_ptr<A>.
}
```

The introduction of `std::auto_ptr` is perhaps the first example of where a user-defined type was added to the standard C++ library in order to define an idiom meant to fix a fundamental C++ language flaw due to incompatible language features. Note the term "flaw" is used and not "deficiency". It is generally excepted in most modern programming languages that the language proper will not support every programming model or idiom that is of general interest and instead (class) libraries are provided to fill in the gaps. The problem is that the language definition itself is flawed with respect to the raw use of `new`/`delete` along with `try`/`throw`/`catch` and is not just simply missing some desirable feature. One could argue that what C++ is really missing is garbage collection (GC) but even that is not the case because to add GC would be fundamentally incompatible with the current user-controlled memory management facility using `new` and `delete`. There is a lot of C++ code out there that requires that destructors for objects be called exactly when expected such as when `delete` is called (and there are idioms such as defined in Section 5.13.4 that depend on this behavior). Any form of language-supported GC will break some backward compatibility of C++ and therefore we may never see a C++ standard with full GC. Also, removing the ability to precisely control when destructors are called and memory is reclaimed would make C++ less attractive for many domains where such low-level control is critical (e.g. embedded programming, systems programming, scientific programming).

The Boost library and the up-coming C++0x standard add more types that continue in this trend of providing new user-defined types and idioms to address fundamental C++ language flaws and deficiencies. However, as described in meat of this paper, both the Boost and the C++0x standard libraries fall short of providing a complete and comprehensive solution to the problems with raw C++ pointers and raw access to memory.

Note that the upcoming C++0x standard as it is currently defined (at least the time of this writing) will do nothing to fix the majority of these nonsensical raw C++ pointer gotchas because to do so would destroy backward compatibility of millions of lines of existing C++ code. Because of the need for backward compatibility, we cannot rely on any future C++ standard to fix the basic problems with raw C++ pointers. Instead, this document advocates using new C++ user-defined types to create a new safer type-system in

C++ and avoiding the direct use of raw C++ pointers except where required to interact with non-compliant code.

# 3 Problems with common approaches for addressing memory management in C++

Because of some of the obvious problems with using raw C++ pointers to access raw memory and using raw calls to `new` and `delete` to perform dynamic memory management, various authors have advocated a number of different approaches for addressing these problems. A few of these approaches will be described along with arguments as to why they are far too sub-optimal.

## 3.1 Problems with using `std::vector` for handling all arrays

A very common approach to try to get around using raw C++ pointers for managing contiguous arrays of data is to use the container class `std::vector` in every use case where a raw C++ array or pointer to an array would be used. Before describing use cases where `std::vector` is being poorly used, first a review is given for what `std::vector` is and what it is good for. The standard library class `std::vector` is a general-purpose concrete contiguous data container class for storing and retrieving value objects[4]. What makes using `std::vector` attractive as compared to a simple class that a developer would write for themselves is that:

- `std::vector` is a Standard Template Library (STL) compliant data container which makes it easy to use with STL-like generic algorithms.
- `std::vector` contains functions for efficiently expanding and shrinking the size of the array that can have platform/compiler specific optimizations with much better performance than what a developer would roll on their own.
- `std::vector` is standardized so one can use it as a means for interoperability with other software in appropriate situations.

These are pretty much the advantages of using `std::vector` over other alternatives. When used as a general purpose data container where one will be changing the size of the array on the fly, `std::vector` is convenient, general, and efficient (just what components from a standard library should be). However, in other use cases, `std::vector` is far from convenient, general, or efficient. As one example, consider using `std::vector` to replace raw C++ pointers for array arguments in all C++ functions as some authors have suggested (e.g. see [21]). For example, consider a VISITOR [17] interface that operates on blocks of data (similar to the RTOp interface described in [6]) along with a concrete subclass shown in Listing 3.

**Listing 3** *:*

```
template<class T>
class BlockTransformerBase {
public:
  virtual ~BlockTransformerBase();
  virtual void transform(const int n, const T a[], T b[]) const = 0;
};
```

[4] See Section 4.1 for a definition of "Value Types".

```
template<class T>
class AddIntoTransformer : public BlockTransformerBase<T> {
public:
  virtual void transform(const int n, const T a[], T b[]) const
    {
      for (int i = 0; i < n; ++i)
        b[i] += a[i];
    }
};
```

The VISITOR interface shown in Listing 3 allows clients to accept any `BlockTransformerBase` object and allow it to transparently implement any number of user-defined transformations. Note that virtual funtions cannot be templated so it is not possible for the `transform(...)` function to be templated on an iterator type but must instead accept some fixed representation of the arrays of data to be operated on. The advantages of the `transform(...)` function in Listing 3 are that a) it is clean, b) the arrays of data can be sub-views of large arrays, and c) it will yield very fast code. Of course the problem with the above function `transform(...)` is that is uses raw C++ pointers. How does the function `transform(...)` know that `a` and `b` are valid pointers and really point to valid arrays of data with at least `n` elements. It is impossible for the function `transform(...)` to assert anything about the data and completely relies on the caller of the function to validate the data. Even in a debug build of the code, there is no way for the implementation of the function `transform(...)` to validate that the preconditions concerning arguments have been met. This is not good and does not allow for even the most basic approaches for defensive programming.

Therefore, some C++ programmers look at this and then they change functions like `transform(...)` in Listing 3 to use `std::vector` which is shown in Listing 4.

**Listing 4** *:*

```
template<class T>
class BlockTransformerBase {
public:
  virtual ~BlockTransformerBase();
  virtual void transform(const std::vector<T> &a, std::vector<T> &b) const = 0;
};

template<class T>
class AddIntoTransformer : public BlockTransformerBase<T> {
public:
  virtual void transform(const std::vector<T> &a, std::vector<T> &b) const
    {
      DEBUG_MODE_ASSERT_EQUALITY( a.size(), b.size() );
      for (int i = 0; i < a.size(); ++i)
        b[i] += a[i];
    }
};
```

The advantages of the function in Listing 4 are that a) the size of each array is kept with the pointer to the array itself inside of each `std::vector` object, b) The sizes of the arrays can be asserted by the

implementation of the function `transform(...)`, c) it is easy for callers who already use single `std::vector` objects.

While this use of `std::vector` replaces raw C++ pointers as basic array function arguments, it has several serious problems in both usability and performance in some important use cases. The primary disadvantages of using `std::vector` as general array arguments to functions is a) there is no flexibility in how the arrays are allocated, and b) one cannot pass sub-views of larger arrays of data.

To illustrate the problems with using `std::vector` for all array arguments to functions, consider a situation where the application wants to allocate big arrays of data and then operate on pieces of the array based on different logic. One motivation for allocating big arrays of data is to avoid memory fragmentation and improve data locality. Now consider in Listing 5 what the client code would have to look like when using the form of `transform(...)` in Listing 4 which takes in `std::vector` objects.

**Listing 5** *: Client code that has to create temporary `std::vector` objects to call function that takes `std::vector` arguments*

```
void someBlockAlgo( const BlockTransformerBase &transfomer,
  const int numBlocks, const std::vector<double> &big_a,
  std::vector<double> &big_b )
{
  DEBUG_MODE_ASSERT_EQUALITY( big_a.size(), big_b.size() );
  const int totalLen = big_a.size();
  const int blockSize = totalLen/numBlocks; // Assume no remainder!

  const int blockOffset = 0;
  for (int block_k = 0; block_k < numBlocks; ++block_k, blockOffset += blockSize)
  {
    if (big_a[blockOffset] > 0.0) {
      // Create temporary std::vectors to do function call
      std::vector a(big_a.begin()+blockOffset,
        big_a.begin()+blockOffset+blockSize);
      std::vector b(big_a.begin()+blockOffset,
        big_b.begin()+blockOffset+blockSize);
      // Do the operation
      transfomer.transform(a, b);
      // Copy back into the output array
      std::copy(b.begin(), b.end(), big_b.begin() + blockOffset);
    }
  }
}
```

As it is clear to see, the above client code that uses the `std::vector` version of `transform(...)` is neither clean, nor efficient since temporary copies of all of the data have to be created just to make the function call and then data has be be copied back into the full array.

Now consider the client code in Listing 6 which uses the raw C++ pointer version of `transform(...)` in Listing 3.

**Listing 6** *: Example driver code that uses the raw-pointer version of `transform(...)`*

```
void someBlockAlgo(const BlockTransformerBase &transformer,
  const int numBlocks, const std::vector<double> &big_a,
  std::vector<double> &big_b )
{
  DEBUG_MODE_ASSERT_EQUALITY( big_a.size(), big_b.size() ); const int
  totalLen = big_a.size(); const int blockSize = totalLen/numBlocks;

  const int blockOffset = 0;
  for (int block_k = 0; block_k < numBlocks; ++block_k, blockOffset += blockSize)
  {
    if (big_a[blockOffset] > 0.0) {
      transformer.transform(blockSize, &big_a[blockOffset], &big_b[blockOffset]);
    }
  }
}
```

As one can clearly see, using the raw C++ pointer version of `transform(...)` makes the client code much cleaning and much more efficient. However, of course, if the client makes any mistakes with its arrays of memory, then the resulting program will yield undefined behavior and (in the best case) will segfault, or will silently produce the wrong result, or (in the worst case) actually produce the right result on the current platform but will fail on other platforms.

The Teuchos memory management array classes make algorithms involving sub-views like shown above very clean, very efficient, and very safe (see the same versions of this example code using these new Teuchos classes in Section 5.5.5).

In summary, `std::vector` is not an efficient or convenient general-purpose replacement for raw C++ pointers as function arguments in many important use cases.

## 3.2 Problems with relying on standard memory checking utilities

Some programmers simply use raw C++ pointers and think that standard memory checking tools like Valgrind[5] and Purify[6] will catch all of their mistakes. When I first started coding in C++ back in 1996, I was very aware of the problems with using raw pointers in C++ after experiencing the segfaults and memory leaks that all C++ programmers experience. At the time, I had experimented some with writing my own utility classes that encapsulated raw C++ pointers and I considered taking that further. However, at that time, I conjectured that going through the effort of encapsulating all raw C++ pointers might be a waste of time because it would not be long until someone came up with a 100% bullet-proof memory checking tool for C++ that would make my feeble programmer-controlled attempts to wrap raw pointers obsolete. After more than 10+ years of C++ programming experience where I have written hundreds of thousands of lines of C++ code on a number of different platforms/compilers, I have come to regret that decision.

Through painful experience and then through some more careful thought, I have come to realize that memory checking tools like Valgrind and Purify will never be able to provide an even sufficient (certainly not 100%) means to validate memory usage in C++ programs. With respect to existing tool implementations, I have experienced cases where both Valgrind and Purify have reported not even a single

[5] `http://valgrind.org`
[6] `http://www.ibm.com/software/awdtools/purify`

warning before the program segfaulted (while running in the tool) with essentially no feedback at all. I will not go into detail about what techniques memory tools like Valgrind and Purify use to verify memory usage other than to say that they can do a lot by just taking control of `malloc(...)` and `free(...)` and in inserting checks into the execution of the program by controlling the manipulation of the program stack.

One such case where Valgrind and Purify were completely unhelpful occurred with an off-by-one error with `std::vector` using Linux/gcc (before I learned the GCC had a checked STL implementation). In the end, the way that I found the off-by-one error was by just staring at the code over and over again until I happened to see the problem. However, what I discovered through two days of debugging was that `std::vector` used its own allocator which allocated big chunks of memory through `malloc(...)`. It then proceeded to do its own memory allocation scheme, which was very fast but was invisible to the watchful eyes of Valgrind and Purify. Any reads to this block of memory looked fine to Valgrind and Purify because it was all contained within the block returned from `malloc(...)`. What the off-by-one error did was to write over a library managed part of the memory block and that silent corruption would doom a later attempt by `std::vector` to allocate memory.

There are other categories of use cases where external memory checking tools like Valgrind and Purify will never be able to verify correct memory usage. One example is semantic off-by-one errors committed in larger blocks of data. To demonstrate this type of error, consider the example code in the function `someBlockAlgo(...)` in Listing 6 which uses the raw C++ pointer version of the function `transform(...)` in Listing 3. Now consider what happens when a developer introduces an off-by-one error such as shown in `transform(...)` in Listing 7.

**Listing 7** *:*

```
template<class T>
void AddIntoTransformer<T>::transform( const int n, const T a[], T b[] )
{
  for (int i = 0; i <= n; ++i)
    b[i] += a[i];
}
```

In case it is not obvious, the off-by-one error shown in Listing 7 is the replacement of the loop termination statement `i < n` with `i <= n` which is a very common C++ programming error.

Now let's consider the implications that the off-by-one error shown in Listing 7 will have on the data in `big_b` as driven by the code in Listing 6. If the last block `block_k=numBlocks-1` of data is processed, then there is a reasonable chance a memory checking tool like Valgrind would catch the off-by-one error being committed at the very end of the array `big_b`. However, as described above, Valgrind may not catch even this type of error. Also, note that turning on bounds checking with `std::vector` (i.e. by enabling `_GLIB_CXX_DEBUG` with gcc) will not catch this error either because of the way the raw pointers are extracted in and and passed in the function call:

```
transformer.transform(blockSize, &big_a[blockOffset], &big_b[blockOffset]);
```

Now consider a defect caused by this off-by-one error for which no automated memory checking tool that will ever be devised will ever be able to catch. This type of defect will occur, for example, when for the last

block `block_k=numBlocks-1` we have `big_a[(numBlocks-2)*blockSize] > 0.0` and `big_a[(numBlocks-1)*blockSize] <= 0.0`. In this case, only the next-to-last block of data will be processed by the defective `transform(...)` function. This will not result in a classic off-by-one error that a memory checking tool would catch because it would not touch memory outside of what is stored in `big_b`. However, this off-by-one error committed in Listing 7 would result in the array entry `big_b[(numBlocks-2)*blockSize+blockSize]` being erroneously modified. This is a defect that might only slightly damage the final result of the program for the typical use case and might therefore go unnoticed for years. However, when the program was really being used for something important years later for a non-typical use case, this small off-by-one error could result in reporting incorrect results with perhaps disastrous consequences.

The point that is trying to be made in the above example is that automated memory checking tools like Valgrind and Purify will never be able to check the *semantic* correctness of the usage of memory. The semantic off-by-one defect described above is 100% correct from a strict memory usage point of view (i.e. only allocated memory can be written to and only allocated and initialized memory can be read from) but is 100% wrong from a semantic point of view (i.e. the function `transform(...)` can only operate on the elements of data from `0` to `n-1`). The array memory management classes in Teuchos described in this document help to verify that memory is used in a semantically correct way and throws exceptions for these types of errors in a debug-mode build.

# 4 Important prerequisites

Before finally discussing the Teuchos memory management classes, a set of prerequisite concepts are presented that are needed in order to understand the holistic memory management approach.

## 4.1 Value types versus reference types

Because of the flexibility of C++, many C++ programmers can and do implement a wide variety of types yielding objects with different types of usage semantics. A quick summary of common "accepted" class types in C++ is given in Item 33 "Be clear what kind of class you're writing" in [31]. There is little point here in trying to classify all of the crazy ways that people have used to code objects in C++ that stray from these "accepted" class types. Instead, the recommendation here is to classify the majority of classes as either *value types* or *reference types*. Value types and reference types are said to use *value semantics* and *reference semantics*, respectively, and that is sometimes how these data-types are described in various C++ literature.

*Value types* in general:

- have public destructors, default constructors, copy constructors, and assignment operators (all implementing deep copy semantics),
- have an identity that is determined by their value not their address,
- are usually allocated on the stack or as direct data members in other class objects,
- are usually not allocated on the heap (but can be for most value-type classes), and
- do not have any virtual functions and are not to be used as base classes (see Item 35 in [31]).

If `S` denotes a typical value type, the class definition of `S` includes:

```
class S {
pubic:
  ~S();
  S();
  S(const S&);
  S& operator=(const S&);
  ...
};
```

All of the built-in intrinsic C++ data-types like `char`, `int`, and `double` are value types. Likewise, class types like `std::complex` and `std::vector` are also value types. Value types have also been called by other names in the C++ literature. Stroustrup refers to value types as "true local variables" in [28]. The term Abstract Data Type (ADT) in older C++ literature such as [9] usually maps to the concept of a value type, but usually carries greater significance in implying that operator overloading is used to make an ADT look more like a built-in C++ type (such as is the case for `std::complex`).

Alternatively, *reference types* in general:

- do not have a public copy constructor or assignment operator,
- are manipulated through a (smart) pointer or reference,
- have an identity that is primarily determined by their address and not their value,
- are allocated on the heap,
- typically are not permitted to be or cannot be allocated on the stack,
- are copied through an abstract clone function (if copying is allowed at all),
- have one or more virtual functions, and
- are usually designed to be used as base classes or are derived from base classes.

Reference types (employing reference semantics) are typically used for base classes in C++. Examples of base classes in the C++ standard library include `std::ios_base` and `std::basic_streambuf`. Reference types in the form of abstract base classes form the foundation for object-oriented programming in C++.

If `A` denotes a typical reference type class, the class definition of `A` generally includes:

```
class A {
pubic:
  virtual ~A();
  virtual A* clone() const = 0;  // NOTE: Should use RCP (see later)
  virtual void someFunc() = 0;
  ...
protected: // or private
  A(const A&);
  A& operator=(const A&);
  ...
};
```

Note that one can almost always choose to manipulate a value type using reference semantics. For example, it is very common to choose to dynamically allocate large value objects like `std::vector` and then pass around (smart) pointers and references to the object to avoid unnecessary and expensive copying and to facilitate the sharing of state.

While the ideas of value types and reference types and value semantics and reference semantics are long established in the C++ literature (even if the terminology is not very uniform), many C++ programmers either seem to not know about these idioms or choose not to follow them for some reason. By forcing the majority of classes into either using *value semantics* or *reference semantics* one eliminates meaningless variability in C++ programs and frees one's self to think about more important things (see the discussion of using standards to actually improve creativity in [24]).

**Side Note:** The somewhat rigid classification of C++ types into value types and reference types is similar in motivation and in many other respects to Eric Evans' differentiation of all domain types into *Value Objects* and *Entities* in Domain Driven Design (DDD) [14]. While there are similarities between DDD's Value Objects and Entities and C++'s value types and reference types, respectively, there is not a one-to-one mapping. In DDD, the distinction between a Value Object and an Entity has more to do with the nature of the object in relation to the domain model and is not related to how memory is manged. Evans assumes that one is using a language like Java where all objects use reference semantics.

## 4.2 Non-persisting versus persisting and semi-persisting associations

Another important prerequisite for understanding the Teuchos memory management classes is the distinction between *non-persisting associations* and *persisting associations*. Working definitions for these are:

- *Non-Persisting associations* are associations between two or more objects that exist only within a single function call for formal function arguments, or a single statement for function return objects, where no memory of any of the objects is retained as a side effect after the function returns or the statement ends.
- *Persisting associations* are associations that exist between two or more objects that extend past a single function call for formal function arguments, or a single statement for function return objects.

To help define these two different types of associations, consider the class and function definitions in Listing 8 and shown in Figure 1.

**Listing 8** *: Classes using raw pointers with both non-persisting and persisting associations*

```
class A {
public:
  void fooA() const;
};

class B {
public:
  void fooB1(const A &a) { a.fooA(); }
  void fooB2() const { ... }
};

class C {
  B* b_;
public:
  C() : b_(0) {}
  void fooC1(B &b, const A &a)
    { b_ = &b; b_->fooB1(A); }
  void fooC2() const
    { b_->fooB2(); }
};

void someFunc(C &c, B &b, const A &a)
{
  c.fooC1(b, a);
  c.fooC2();
}
```

The function `B::fooB1(...)` in Listing 8 involves a non-persisting association with respect to the `A` and `B` objects since no memory of the object `a` remains after the function `B::fooB1(...)` exists. Non-persisting associations represent typical input/output-only arguments to a function.

The function `C::fooC1(...)` in Listing 8 creates a persisting association between a `C` object and a `B` object since the memory of the `B` object is retained in the `C` object that persists after the function `C::fooC1(...)` exits. This memory of the `B` object stored in the `C::b_` pointer data member is then used to implement the function `C::fooC2()`. Note that the function `C::fooC1(...)` also involves a non-persisting association with the `A` object `a` since it is only used to call `B::fooB1(...)` and no memory of `a` lingers after `C::fooC1(...)` exists.

Another interesting case is the nonmember function `someFunc(...)` also shown in Listing 8. While `someFunc(...)` is a free function, it actually involves the creation of a persisting association between the `C` and `B` objects as a side effect because it calls the `C::fooC1(...)` function.

In the idioms advocated in this paper, smart reference counted pointers are used for all persisting associations and never for non-persisting associations. Using the basic Teuchos `RCP` class, the raw pointer code in Listing 8 would be refactored into the code shown in Listing 9.

**Listing 9** *: Refactored classes to use `RCP` for persisting associations*

```
class A {
public:
  void fooA() const;
};

class B {
public:
  void fooB1(const A &a) { a.fooA(); }
  void fooB2() const { ... }
};

class C {
  RCP<B> b_;
public:
  void fooC1(const RCP<B> &b, const A &a)
    { b_ = b; b_->fooB1(A); }
  void fooC2() const
    { b_->fooB2(); }
};

void someFunc(C &c, const RCP<B> &b, const A &a)
{
  c.fooC1(b, a);
  c.fooC2();
}
```

Note that the classes `A` and `B` remain unchanged because they do not involve any persisting associations. It is only the class `C` that needed to be refactored to handle the persisting association with `B`. The non-member function `someFunc(...)` is also be modified since the creation of a persisting association is involved.

Most programming languages do not provide any means to differentiate between non-persisting associations and persisting associations (see Section 6.1 for an expanded discussion). However, note that the Unified Modeling Language (UML [16]) does differentiate between them in that persisting

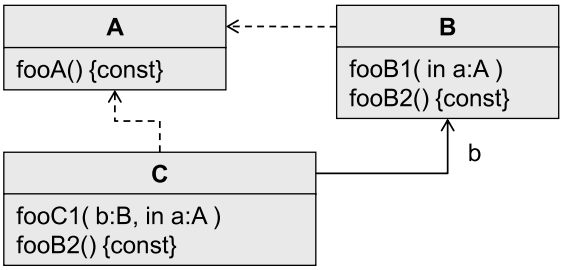



**Figure 1.** UML Class Diagram showing non-persisting and persisting associations for classes in Listing 8 and Listing 9.

relationships are shown with a solid line while non-persisting relationships are shown with a dotted line as depicted for these example classes in Figure 1.

Another situation where the concepts of persisting and non-persisting associations comes up relates to how objects are returned by C++ functions as return values. A persisting relationship is created through a function return object if that object is remembered past a single statement. For example, consider the code fragment in Listing 10.

**Listing 10** *: Example of a dangerous type of persisting association that can result in undefined behavior (e.g. segfault)*

```
std::vector<int> v(n);
...
int &ele = v[0];  // Creates a persisting return object relationship
...
ele = 5;          // Changes v[0] much later!
```

The code fragment in Listing 10 shows a presenting relationship between the client code that is initializing the local reference `ele` and the `std::vector` container object `v`. This is very fragile and dangerous code because if `v` is re-sized, grown or have some other type of change, then the reference pointed to by `ele` can become invalid. For example, the following code fragment will likely result in a runtime memory usage error with undefined behavior and (if one is lucky) will segfault:

**Listing 11** *:*

```
std::vector<int> v(n);
int &ele = v[0];
v.resize(10*n);
ele = 5;  // ele is likely to be invalid here!
```

If one is unlucky, the statement `ele = 5` will work just fine on one platform with one implementation of the `std::vector` but will break on another platform when run with a different data set. Note that tools like

Valgrind and Purify may not flag the problem due to the way that many implementations of `std::vector` deal with memory.

Basically the problem here is that the `std::vector::operator[](size_type)` function returns a raw C++ reference that should never be remembered past a single statement. The safe way to change an element is:

**Listing 12** *:*

```
std::vector<int> v(n);
v[0] = 5; // Non-persisting relationship!
v.resize(10*n);
```

Therefore, functions like `std::vector::operator[](size_type)` which return raw C++ references should only be used for non-persisting associations as shown above.

Before leaving the topic of persisting and non-persisting associations, one has to recognize that there exists a third category of associations that lie in between strict persisting and non-persisting associations. This gray area will be referred to here as semi-persisting associations defined as:

- *Semi-persisting associations* are associations that (like persisting associations) exist between two or more objects that extend past a single function call for formal function arguments, or a single statement for function return objects except where the use of the objects and the lifetime of the association have more rigid constraints requiring greater care in use.

An example of a semi-persisting association is the use of an iterator to access an STL container as shown in Listing 13:

**Listing 13** *:*

```
void someFunc(std::vector<int> &v)
{
  typedef std::vector<int>::iterator itr_t;
  for (itr_t itr = v.begin(); itr != v.end(); ++itr) {
    *itr = 5;
  }
}
```

As shown in Listing 13, the iterator object `itr` is used well past (perhaps thousands of loop iterations) where it was created by the statement `itr_t itr = v.begin()`. There are, however, significant restrictions on how such iterators can be used: a) the iterator cannot be accessed after the originating parent object has been destroyed, and b) the iterator cannot be accessed after the structure of the originating parent has changed (e.g. `v.resize(...)` was called). For the sake of performance, one has to allow for the use of semi-persisting associations such as this where the optimized build of the code lacks the machinery to detect invalid usage. However, note that in the case of the STL containers that in a debug-mode checked

STL build (supported by GCC and the Microsoft compiles for instance), these types of dangling iterator references will typically be safely detected. This type of debug-mode runtime checking is the saving grace for the use of iterators and other types of semi-persisting associations which makes their use acceptable. If iterators to `std::vector` where simply hard-coded as raw pointers, this type of debug-mode runtime checking would not be possible.

Semi-persisting associations also play a role in the use of the Teuchos memory management classes in situations where performance is critical (see Section 5.12.3 and Commandments 6 and 8 in Appendix B).

# 5 Teuchos classes for safer memory management and usage

The primary purpose for the Teuchos memory management classes is to encapsulate all raw C++ pointers in all high-level code and eliminate undefined behavior in C++ programs without sacrificing portability, control or efficiency. These classes are efficient and general and, in a debug-mode build of the code, will catch and gracefully report 99% or so of the programming errors typically made with the ubiquitous high-level use of raw C++ pointers.

## 5.1 Overview of basic approach employed by Teuchos memory management classes

The basic approach being advocated here and implemented in the Teuchos memory management classes is to:

- Encapsulate all raw C++ pointers in high-level code using specially designed memory management classes, capture raw C++ pointers as soon as possible, and encapsulate raw calls to `new` in library and application code.
- Provide a complete set of cooperating types that work together to safely and conveniently implement all hand-offs of encapsulated raw C++ pointers using carefully scrutinized conversion code provided with the classes. Also, never define implicit conversions from these safe types to raw C++ pointers (or the safety of the entire type system falls apart).
- Differentiate memory management classes for handling single objects from those for handling contiguous arrays of objects.
- Differentiate memory management classes according to persisting and non-persisting (and semi-persisting) associations.
  - Use reference counting for memory management classes designed to handle persisting associations.
  - Do not impose the overhead of reference counting for memory management classes designed to handle non-persisting associations.
  - Do not impose the overhead of reference counting for memory management classes designed to handle semi-persisting associations (but provide the machinery for strong debug-mode runtime checking).
- Provide encapsulations for all uses of raw C++ pointers for arrays of contiguous objects including dynamically sized arrays, statically sized arrays, and stack-based arrays.
- Provide a default *optimized mode* where maximum performance and minimal overhead are the goals where raw C++ pointer performance is achieved for all reasonable use cases.
- Provide an optional *debug mode* whose goal is to provide maximum runtime checking with reasonably low overhead in order to catch all sorts of common errors like:
  - Dereferencing null pointers (Section 5.11.1)
  - Array access errors like off-by-one and other errors (Section 5.11.1)

**Basic Teuchos smart pointer types**

| | Non-persisting (and semi-persisting) Associations | Persisting Associations |
|---|---|---|
| single objects | `Ptr<T>` | `RCP<T>` |
| contiguous arrays | `ArrayView<T>` | `ArrayRCP<T>` |

**Table 1.** Basic Teuchos memory management utility classes for encapsulating raw pointers.

**Summary of operations supported by the basic Teuchos smart pointer types**

| Operation | `Ptr<T>` | `RCP<T>` | `ArrayView<T>` | `ArrayRCP<T>` |
|---|---|---|---|---|
| *Raw pointer-like functionality* | | | | |
| Implicit conv derived to base | x | x | | |
| Implicit conv non-const to const | x | x | x | x |
| Dereference `operator*()` | x | x | | x |
| Member access `operator->()` | x | x | | x |
| `operator[](i)` | | | x | x |
| operators `++`, `--`, `+=(i)`, `-=(i)` | | | | x |
| *Other functionality* | | | | |
| Reference counting machinery | | x | | x |
| Iterators: begin(), end() | | | x | x |
| `ArrayView` subviews | | | x | x |

**Table 2.** Summary of capabilities of the basic Teuchos memory management classes.

  - Incorrect iterator usage (Section 5.11.1)
  - Circular dependencies (Section 5.11.2)
  - Dereferencing dangling pointers (references) (Section 5.11.3)
  - Multiple owning reference-counting node object (Section 5.11.4)

- Structure debug-mode checking such that it does not alter the observable behavior of correct programs in any way. However, when debug-mode checking is enabled, the software should never yield undefined behavior (e.g. segfault).

The basic templated Teuchos memory management classes for encapsulating raw C++ pointers for single objects and arrays are `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` shown in Table 1. A summary of the capabilities of these classes is shown in Table 2. What one can see from this table is that raw pointer-like functionality is partitioned across these various sets of classes in logical and safe ways. For example, array-related operations are not defined on the single-object classes `Ptr` and `RCP` but implicit conversion from derived types to base types is allowed. Alternatively, the array classes `ArrayView`, and `ArrayRCP` do not support the dangerous and ill-conceived ability to implicitly convert arrays of derived types to arrays of

**Other Teuchos array container classes**

| Array class | Specific use case |
|---|---|
| `Array<T>` | Contiguous dynamically sizable, expandable, and contractible arrays |
| `Tuple<T,N>` | Contiguous statically sized (with size `N`) arrays |

**Table 3.** Teuchos array container classes.

base types that is discussed in Section 2.2. Note that the class `ArrayView` does not support all of the raw pointer iterator-like operations that `ArrayRCP` supports like the dereference operator `operator*()` or the pointer offset functions that change the pointer. The reason that `ArrayRCP` does support these iterator-like operations is so that it can be used as a general purpose iterator implementation while `ArrayView` objects do not need to be used in this way. Note that all of the array classes defined in Teuchos all support a basic iterator interface with the `[const]_iterator` member typedefs and the functions `begin()` and `end()`. In optimized mode, these iterators are simply raw C++ pointers yielding maximum performance. However, in debug mode, the iterators are implemented as `ArrayRCP` objects resulting in fully checked iterators.

In addition to the four basic memory management classes shown in Table 1 (which provide the most fundamental encapsulations for all raw C++ pointers in all high-level code) Teuchos also contains a few other array container classes for a few more specific use cases shown in Table 3. The array container classes `Array` and `Tuple` pretty much cover the majority of use cases in C++ where raw C++ pointer arrays are used for containers. The class `Array` is a direct replacement for `std::vector` and actually wraps it internally.

Note that all of these classes are templated on value types and are themselves value-types (see Section 4.1). This means that one can embed these types in each other in any arbitrary order to create any type of data structure that one would like. For example, one could have `Array<RCP<ArrayRCP<ArrayView<Tuple<Ptr<T>,5> > > > > >`. By understanding what each of these types provide and what each type means (in terms of the idioms defined in Section 5.8), one can achieve almost anything in a way that is self documenting.

These classes all work together to provide a high level of debug-mode runtime checking to catch the majority of common programming errors and report these errors gracefully with informative error messages. A debug-mode build of the code is facilitated by defining the preprocessor macro `TEUCHOS_DEBUG` (through the CMake configuration variable `Teuchos_ENABLE_DEBUG=ON`). When `TEUCHOS_DEBUG` is not defined, the Teuchos memory management classes are configured to impart minimal overhead and yield fast performance. When `TEUCHOS_DEBUG` is defined, these classes are configured to perform maximal debug runtime checking. These classes are also carefully designed so that if a program is implemented correctly using these classes and executes correctly in optimized mode, then the program compiled with the debug checking turned on will execute in exactly the same way. However, if any undefined, dangerous, or just plain wrong behavior is being used, then these memory management classes will throw exceptions and the exception objects will have very good error messages embedded in them making it easier to debug and fix the problems.

What is important to understand is that all of these memory management classes must be developed together with knowledge of each other's internal implementations in order to provide solid debug-mode runtime checking. For example, in general, one cannot mix together different memory management classes

like `boost::shared_ptr` (i.e. `std::shared_ptr` in C++0x) and `std::vector` with the Teuchos memory management classes at the user level and provide the same high level of runtime checking that is supported by the complete system of Teuchos memory management classes. More details about why this is so are given in Section 5.11 in the context of debug-mode runtime checking for and reporting of dangling references.

As with the development of any set of C++ classes, a set of accompanying idioms must also be developed for maximizing their effective use. The idioms described in this paper involving the Teuchos memory management classes result in code with maximum compile-time checking, maximum debug-mode runtime checking, near optimal perforamnce, and maximally self-documenting.

## 5.2 The proper role of raw C++ pointers

The main thesis of this paper is that the use of all raw C++ pointers should be fully encapsulated in all high-level C++ code and instead a system of safe specially designed types tailored to specific use cases should be used instead. Does that mean that raw C++ pointers should never be used in any C++ code? The answer of course is no, since raw C++ pointers will always have to be used in some special situations (but perhaps not directly used as described below).

Given the full use of the Teuchos memory management classes, here are the valid situations where it is appropriate (or required) to use raw C++ pointers in fully compliant C++ programs:

- *Use raw C++ pointers (indirectly) for extremely well-encapsulated, low-level, high-performance algorithms*

  In order to achieve high performance in computationally intensive code, one will always have to use raw C++ pointers (at least indirectly) in a non-debug optimized build. This includes using raw C++ pointers to build specialized data structures and similar purposes. In this context, one can think of raw C++ pointers as a fairly compact and efficient way to communicate with the compiler about how one wants to manage memory at the hardware level. The ability to do this type of fine-grained manipulation of memory has always been one of the strengths of C and C++ in systems-level programming. Therefore, one can think of using raw pointers in C++ as a kind of portable assembly language that one always has at one's disposal on every platform and with every compiler. However, instead of using raw pointers, one can always use the types `Ptr`, `ArrayView` or `ArrayType::iterator` (where `ArrayType` is `Array`, `ArrayRCP`, or `ArrayView`) to yield raw pointer performance in a non-debug optimized build but still maintain strong debug-mode runtime checking. This approach is discuss in more detail in Section 5.12.3.

- *Use raw C++ pointers to communicate with legacy C++ code and with other languages through C bindings*

  The only remaining valid reason to use raw C++ pointers is to reuse and communicate with legacy C++ code and to call functions in other languages through the now-universal approach of using C bindings. However, one must endeavor to minimize the amount of C++ code that has naked raw C++ pointers and one should only expose a raw C++ pointer at the last possible moment (such as in the call to the external functions themselves). Again, one must carefully encapsulate access to non-compliant code that requires the exposure of raw C++ pointers.

| **Common member functions** |
|---|
| `T* getRawPtr() [const]` |
| **Common nonmember functions** |
| `void swap(Type<T>&, Type<T>&)` |
| `bool is_null(const Type<T>&)` |
| `bool nonnull(const Type<T>&)` |
| `bool operator==(const Type<T>&, ENull)` |
| `bool operator!=(const Type<T>&, ENull)` |
| `bool operator==(const Type<T>&, const Type<T>&)` |
| `bool operator!=(const Type<T>&, const Type<T>&)` |
| `bool operator<(const Type<T>&, const Type<T>&)` |
| `bool operator<=(const Type<T>&, const Type<T>&)` |
| `bool operator>(const Type<T>&, const Type<T>&)` |
| `bool operator>=(const Type<T>&, const Type<T>&)` |

**Table 4.** Common members and non-members for `Type` = `Ptr`, `RCP`, `Array` `ArrayRCP`, `ArrayView`, and `Tuple`.

One point is worth nothing here which is that in this new modern C++ software one must never use raw C++ pointers in the basic interfaces between the various modules as that is where a majority of mistakes in the use of memory will be made. This goes somewhat contrary to the advice in Item 63 "Use sufficiently portable types in a module's interface" in [31]. If this new modern safe C++ software must be called by non-compliant software that uses raw C++ pointers, then one can provide specialized C-like interfaces for those clients that use raw C++ pointers for communication. Of course, once one does this, one will have to rely on clients to pass in memory correctly and keep it valid as long as local modules need it.

### 5.3 Common aspects of all Teuchos memory management classes

Table 4 gives the member and non-member functions common all the Teuchos memory management classes `Ptr`, `RCP`, `Array`, `ArrayView`, `ArrayRCP`, and `Tuple`. The comparison operators allows all of these types to be used as keys in associative containers like `std::map`. The member function `getRawPtr()` is actually an incredibly useful function that will return a null pointer (i.e. 0) when the underlying smart pointer is null (such as with `Ptr`, `RCP`, `ArrayRCP`, and `ArrayView`) or when the container has size zero (such as with `Array`).

### 5.4 Memory management classes replacing raw pointers for single objects

The templated classes `Ptr` and `RCP` described in the next two sections are used to encapsulate raw C++ pointers to single objects. Again, `Ptr` is used for non-persisting (and semi-persisting) associations and `RCP` is used for persisting associations. Below, and in all of the code listings, it is assumed that the code is enclosed in the `Teuchos` namespace or there are appropriate `using Teuchos::XXX` declarations (as is safe and appropriate) for the various names in place.

### 5.4.1 `Teuchos::Ptr<T>`

The templated class `Ptr` is the simplest of all the Teuchos memory management classes. In optimized mode it is just the thinnest of wrappers around a raw C++ pointer. Listing 14 shows what the implementation of `Ptr` looks like in optimized mode:

**Listing 14** *: Teuchos::Ptr class*

```
template<class T>
class Ptr {
public:
  Ptr( ENull null_in = null ) : ptr_(0) {}
  explicit Ptr( T *ptr ) : ptr_(ptr) {}
  Ptr(const Ptr<T>& ptr) : ptr_(ptr.ptr_) {}
  template<class T2> Ptr(const Ptr<T2>& ptr) : ptr_(ptr.ptr_) {}
  Ptr<T>& operator=(const Ptr<T>& ptr) { ptr_=ptr.ptr_; return *this; }
  T* operator->() const { return ptr_; }
  T& operator*() const { return *ptr_; }
  T* getRawPtr() const { return ptr_; }
  T* get() const { return ptr_; } // For compatibility with shared_ptr
  const Ptr<T>& assert_not_null() const;
private:
  T *ptr_;
};
```

In optimized mode, the only overhead imparted by `Ptr` is the default initialization to null (0). All other functions are just inline accessors to the underlying raw C++ pointer member `ptr_`. Therefore, the performance when using this type is the same as when using a raw C++ pointer.

However, in debug mode (enabled when `TEUCHOS_DEBUG` is defined), the `Ptr` class becomes more complex and performs a number of runtime checks like for null dereferences and dangling references (see Section 5.11.3).

One note about the default null constructor shown in Listing 14 which is:

```
template<class T>
Ptr<T>::Ptr( ENull null_in = null ) : ptr_(0) {}
```

is that the type `ENull` is the simple enum in the `Teuchos` namespace:

```
enum ENull { null };
```

This simple enum allows for the safe implicit conversion from the enum value `null` to any `Ptr<T>` object. For example, one can write code like:

```
Ptr<A> a_ptr = null;
```

This implicit conversion from `null` is shared by the other Teuchos memory management smart-pointer classes `RCP<T>`, `ArrayView<T>`, and `ArrayRCP<T>`. This allows calling functions that accept one of these objects and by just passing in `null` when appropriate and the implicit conversion will be done automatically if possible (see Section 5.7.3).

The main purpose for the existence of the `Ptr` class is to replace raw C++ pointers in function calls for typical input, input/output, and output arguments where no persisting relationship is present. (the class `Ptr` should also be used for semi-persisting associations where single objects are involved.) For example, consider the function that modifies a type `A` object shown in Listing 15.

**Listing 15** *: Simple function using unsafe raw pointer*

```
void modifyA( A *a )
{
  assert(a);
  a->increment();
}
```

Using `Ptr`, the function `modifyA(...)` in Listing 15 would be changed to the form shown in Listing 16.

**Listing 16** *: Simple function refactored to use safe `Ptr` wrapped pointer*

```
void modifyA( const Ptr<A> &a )
{
  a->increment();
}
```

In this context, the primary advantage of the form shown in Listing 16 as apposed to Listing 15 is that in debug mode, a check for a null pointer or a dangling reference would be performed automatically. If a null dereference occurred, then an exception would be thrown with a very good error message. I have seen platforms where a null dereference did not automatically result in a graceful assert, stopping the program. I have seen cases where somehow memory was corrupted and the program continued! A good philosophy is to make as few assumptions as possible about undefined behavior of the compiler and platform because I have found that "typical and obvious" behavior for undefined behavior is not universal. Many have learned the hard way that one will pay a price for such assumptions in lost time debugging obscure things like a null pointer dereference that should have stopped the program but did not. Don't take chances with undefined behavior in the code, take control!

When all of the high-level code has been converted over to use these memory management classes and there are no more raw C++ pointers, then client code should never have to construct a `Ptr` object using a raw C++ pointer. However, as code is being transitioned over and when such code is called by non-compliant code, construction from a raw pointer is needed. The recommended way to convert from a raw C++ pointer to `Ptr` is to use the following templated non-member function:

**Listing 17** *: Teuchos::ptr(...)*

```
template<class T>  Ptr<T> ptr(T *p);
```

Using this non-member constructor function, client code would then be written as shown in Listing 18.

**Listing 18** *:*

```
void foo( A* a )
{
  using Teuchos::ptr;
  modifyA(ptr(a));
}
```

A more typical use case for the construction of a `Ptr` object is from a raw C++ object or reference. This type of construction should always be performed using one of the non-member constructor functions shown in Listing 19.

**Listing 19** *: Safe nonmember constructors for Teuchos::Ptr*

```
template<typename T> Ptr<T>        ptrFromRef( T& arg );
template<typename T> Ptr<T>        inOutArg( T& arg );
template<typename T> Ptr<T>        outArg( T& arg );
template<typename T> Ptr<T>        optInArg( T& arg );
template<typename T> Ptr<const T>  constOptInArg( T& arg );
```

The different forms of non-member constructor functions shown in Listing 19 are to allow for self-documenting code for calls to functions that accept `Ptr`-wrapped objects. A complete and comprehensive set of idioms for using `Ptr` along with the other Teuchos memory management types is given in Section 5.8.

### 5.4.2 `Teuchos::RCP<T>`

The class `RCP`, the real workhorse of the Teuchos memory management classes, is used to manage single objects in persisting associations. `RCP` is very similar to other high-quality reference-counted smart pointer classes like `boost::shared_ptr` and of course the upcoming standard C++0x class `std::shared_ptr`. However, `RCP` has some key features that differentiate it from these other better known smart pointer classes. In particular, `RCP` has built in support for the detection of circular references (Section 5.9.2), has built in support for resolving circular references with built-in weak pointers (Section 5.9.1), and other strong debug runtime checking such as detecting multiple non-related `RCP` objects owning the same reference-counted objects (Section 5.11.4) and other types of checks.

Because the class `RCP` is described in [2] and is so similar in use to `boost::shared_ptr` (described some in [26]), this class will not be described in too much detail here. However, a fairly complete definition of the class `RCP` is shown in Listing 20 (the full listing can be found in the Doxygen documentation).

**Listing 20** *: Class and helper function listing for `RCP`*

```
template<class T>
class RCP {
```

```
public:

  // General functions
  RCP(ENull null_arg = null);
  explicit RCP(T* p, bool has_ownership = false);
  template<class Dealloc_T> RCP(T* p, Dealloc_T dealloc, bool has_ownership);
  RCP(const RCP<T>& r_ptr);
  template<class T2> RCP(const RCP<T2>& r_ptr);
  ~RCP();
  RCP<T>& operator=(const RCP<T>& r_ptr);
  bool is_null() const;
  T* operator->() const;
  T& operator*() const;
  T* getRawPtr() const;
  Ptr<T> ptr() const;

  // Other shared_ptr compariblity functions
  ...

  // Reference counting member functions
  ...

private:
  T *ptr_;
  RCPNodeHandle node_;
  ...
};

// General non-member constructor functions
template<class T> RCP<T> rcp(T* p, bool owns_mem = true);
template<class T> RCP<T> rcpFromRef(T& r);
template<class T> RCP<T> rcpFromUndefRef(T& r);

// Deallocation policy functions
...

// Embedded objects functions
...

// Extra data functions
...

// Conversion functions
...

// Other common non-member functions
...
```

Again, basic usage of the `RCP` class is described in [2] and the functions for decallocation policies, embedded objects, extra data, conversion functions and other functions are discussed in other sections in a more general setting. The basic idioms for smart pointers and reference counting are fairly well known, are well documented in the literature, and there is a good overview in [2] so basic information will not be

replicated here. However, some of the more advanced functionality for `RCP` that is not described in [2] or an any of the existing C++ literature is described in later sections of this document.

### 5.4.3 Raw C++ references

Why is there a subsection on raw C++ references under the a section describing Teuchos Memory Management classes for single objects? The reason is that raw C++ references to single objects are used in the idioms described in this paper for non-persisting associations for single objects and this was a reasonable place to discuss issues with raw C++ references.

While this paper argues that raw C++ pointers have no place in application-level code because they are fundamentally unsafe, are C++ references also not inherently unsafe as well? After all, under the covers raw C++ references really are just raw C++ pointers in disguise. While this is true, in practice raw C++ references are significantly safer than raw C++ pointers, especially if the idioms outlined in this paper are carefully followed. In addition, the use of raw C++ references is exploited (as explained in Section 5.4.3) in defining idioms that increase the self-documenting nature of C++ code and play a role in defining non-persisting associations related to function formal arguments and return objects. All in all, the increased expressiveness in using raw C++ references is worth the increased risk of misuse (this is still going to be C++ after all).

Basically, a raw C++ reference is relatively safe as long as a) it is always initialized to point to a valid object, and b) it is only used for non-persisting relationships (e.g. as const input arguments in C++ functions). If a raw C++ reference is initialized directly from an object or from dereferencing a smart pointer, then it is guaranteed that the object will be valid when the reference is first created (at least in a debug build where dereferencing null smart pointers throws). While raw C++ references are fairly safe when used with the idioms described in this paper, there are no 100% guarantees. There are typically no guarantees that the object pointed to by a raw reference will stay valid (because dangling references cannot be detected as described in Section 5.11.6). This can happen when one breaks one or more of the idioms or guidelines defined in this paper (which will happen because programmers make mistakes).

Note that raw C++ references should never be used for representing semi-persisting associations because it is impossible to catch invalid usage like dangling references. Instead, when a semi-persisting association is involved, always use `Ptr` instead of a raw C++ reference (even if the object being represented is not allowed to be null). Semi-persisting associations are described in more detail in Section 4.2 and Section 5.12.3.

## 5.5 Memory management classes replacing raw pointers for arrays of objects

The Teuchos memory management module actually defines four different C++ classes for dealing with contiguous arrays of objects: `ArrayView`, `ArrayRCP`, `Array`, and `Tuple`. As stated in Section 5.1 each of these classes is needed in order to address different important use cases for dealing with contiguous arrays of objects. The conventions outlined in the paper never have high-level code exposing a raw C++ pointer to an array or directly using built-in (statically sized) C++ arrays.

In addition to the common members shown in Table 4, all of the Teuchos array classes provide a common subset of the interface of `std::vector` which includes the typedefs and member functions shown in Table 5.

| **std::vector compatible member typedefs** |
|---|
| `value_type` |
| `size_type` |
| `difference_type` |
| `pointer` |
| `const_pointer` |
| `reference` |
| `const_reference` |
| `iterator` |
| `const_iterator` |
| `element_type` |
| **std::vector compatible member functions** |
| `size_type size()` |
| `[const_]reference operator(size_type) [const]` |
| `[const_]reference front() const` |
| `[const_]reference back() const` |
| `[const_]iterator begin() [const]` |
| `[const_]iterator end() [const]` |
| **ArrayView returning member functions** |
| `ArrayView<[const] T> view(size_type offset, size_type size) [const]` |
| `ArrayView<[const] T> operator[]()(size_type offset, size_type size) [const]` |
| `ArrayView<[const] T> operator()() [const]` |
| **Additional common member functions** |
| `[const_]pointer getRawPtr() [const]` |
| `std::string toString() const` |

**Table 5.** Additional common members and non-members for `ArrayView`, `ArrayRCP`, `Array`, and `Tuple` .

A few things to note about the common array interface components shown in Table 5 include:

- All of the Teuchos array classes are drop-in replacements for any code that uses `std::vector` that does not grow or shrink the container by supporting the necessary typedefs, query functions, element access, and iterator access. This helps in migrating current code that uses `std::vector` but should be using `Array`, `ArrayView`, `ArrayRCP` or `Tuple`.
- All of the array classes support returning `ArrayView` subviews of contiguous ranges of elements.
- All of the array classes support a handy `getRawPtr()` function that allows a client to get the base pointer address to the array or null. The standard `std::vector` class supports no such function which is very painful for users since it makes it hard to get a null pointer then the container can legitimately be unsized in some use cases.

The exact functions shown in Table 5 for `ArrayView` and `ArrayRCP` are a little different than for `Array` due to the different nature of these view classes as apposed to the container class `Array`. As described in Section 5.6, the classes `ArrayView` and `ArrayRCP` can encapsulate both non-const and const types `T` as their template argument while `Array` can only accept a non-const type `T`. Therefore, the `std::vector` compatible functions in `ArrayView` and `ArrayRCP` are all `const` functions since they don't change what data these objects point to, but only change the data itself.

One other aspect to note about the Teuchos array classes is that they deviate from the standard C++ library convention of using an unsigned integer for `size_type`. Instead, they use a signed integer for `size_type` typedefed to the signed type `Teuchos_Ordinal` which is guaranteed to be 32 bit on a 32 bit machine and 64 bit on a 64 bit machine[7]. The reasoning for breaking from the `std::vector` standard for `size_type` is described in Appendix C.

### 5.5.1 `Teuchos::ArrayView<T>`

The class `ArrayView`, the simplest of the Teuchos array memory management classes, is designed to encapsulate raw pointers in non-persisting associations primarily for formal function array arguments. (`ArrayView` is to be used for semi-persisting associations as well.) In an optimized build, an `ArrayView` object simply holds a raw base array pointer and an integer size. In an optimized build, `ArrayView` looks like Listing 21.

**Listing 21** *: `Teuchos::ArrayView` declaration (See Table 5 for common array members.)*

```
template<class T>
class ArrayView {
public:

  // Constructors/Assignment/Destructors
  ArrayView( ENull null_arg = null );
```

[7] `Teuchos_Ordinal` is typedefed by default to the standard C library type `ptrdiff_t` which is always signed and is 32 bit on a 32 bit machine and 64 bit on a 64 bit machine.

```
  ArrayView( T* p, size_type size );
  ArrayView(const ArrayView<T>& array);
  ArrayView(std::vector<typename ConstTypeTraits<T>::NonConstType>& vec);
  ArrayView(const std::vector<typename ConstTypeTraits<T>::NonConstType>& vec);
  ArrayView<T>& operator=(const ArrayView<T>& array);
  ~ArrayView();

  // Implicit conversion to const
  operator ArrayView<const T>() const;

  // Deep copy
  void assign(const ArrayView<const T>& array) const;

  // Common array class members and other functions
  ...

private:
  T *ptr_;     // Optimized implementation
  size_type size_;

};

// Non-member helpers

template<class T>
ArrayView<T> arrayView( T* p, typename ArrayView<T>::size_type size );

template<class T>
ArrayView<T> arrayViewFromVector( std::vector<T>& vec );

template<class T>
ArrayView<const T> arrayViewFromVector( const std::vector<T>& vec );

template<class T>
std::vector<T> createVector( const ArrayView<T> &av );

template<class T>
std::vector<T> createVector( const ArrayView<const T> &av );

// Other common non-member helpers
...

// Explicit conversion functions
...
```

A few specific things to note about `ArrayView` shown in Listing 21 in addition to the comments in Section 5.5 and other sections include:

- `ArrayView` is extremely lightweight in an optimized build, carrying only a pointer and an integer size. This allows one to replace the typical pointer and separate size argument with a single aggregate light-weight object. Therefore, it yields very efficient code.

- `ArrayView` in optimized mode has all trivial inlined functions that work with the raw pointer so it is as efficient as raw pointer code (Section 5.12.2).
- `ArrayView` is a drop in replacement for any code that uses `std::vector` that does not grow or shrink the container by supporting the necessary typedefs, query functions, and iterator access. This helps in migrating current code that uses `std::vector` but should be using `ArrayView`.
- `ArrayView` implicitly converts from an `std::vector` so functions called by existing client code that uses `std::vector` can be safely and transparently refactored to use `ArrayView` instead of `std::vector` (subject to the limitations for implicit conversions described in Section 5.7.3).
- `ArrayView` directly supports the creation of subviews of contiguous ranges of elements.

What makes `ArrayView` non-trivial and special, however, is that in a debug build, the implementation takes on a variety of runtime checking to catch all sorts of errors such as dangling iterators, dangling sub-views (Section 5.11.3), range checking (Section 5.11.1), and other types of runtime checking.

It should be noted that one should almost never create an `ArrayView` object directly from a raw pointer but instead create them as views of `Array`, `ArrayRCP`, `Tuple` and other `ArrayView` objects. If client code is routinely creating `ArrayView` objects from raw pointers, then the code is not safe and one needs to study the core idioms described in Section 5.8.

The class `ArrayView` has no equivalent in boost or the current C++ or proposed C++0x standard. This is a critical class needed to allow for flexibility, high-performance, safety, and maximaly self-documenting code. One cannot develop an effective type system without an integrated type like `ArrayView`.

### 5.5.2 `Teuchos::ArrayRCP<T>`

The class `ArrayRCP` is the counterpart to `ArrayView` for general flexible array views except it is used for persisting relationships where reference-counting machinery is required. An `ArrayRCP` object can provide a contiguous view into any array of data allocated in anyway possible and can allow the user to define what is done to release memory in anyway they would like.

The class declaration for `Teuchos::ArrayRCP` is shown in Listing 22.

**Listing 22** *: `Teuchos::ArrayRCP` declaration (optimized build)*

```
template<class T>
class ArrayRCP {
public:

  // Constructors/initializers
  ArrayRCP(ENull null_arg=null);
  ArrayRCP(T* p, size_type lowerOffset, size_type upperOffset,
    bool has_ownership);
  template<class Dealloc_T>
    ArrayRCP( T* p, size_type lowerOffset, size_type upperOffset,
      Dealloc_T dealloc, bool has_ownership);
  explicit ArrayRCP(size_type lowerOffset, const T& val = T());
```

```
  ArrayRCP(const ArrayRCP<T>& r_ptr);
  ~ArrayRCP();
  ArrayRCP<T>& operator=(const ArrayRCP<T>& r_ptr);

  // Object/Pointer Access Functions
  T* operator->() const;
  T& operator*() const;
  ArrayRCP<T>& operator++();
  ArrayRCP<T> operator++(int);
  ArrayRCP<T>& operator--();
  ArrayRCP<T> operator--(int);
  ArrayRCP<T>& operator+=(size_type offset);
  ArrayRCP<T>& operator-=(size_type offset);
  ArrayRCP<T> operator+(size_type offset) const;
  ArrayRCP<T> operator-(size_type offset) const;

  // ArrayRCP Views
  ArrayRCP<const T> getConst() const;
  ArrayRCP<T> persistingView(size_type lowerOffset, size_type size) const;

  // Implicit conversions
  operator ArrayRCP<const T>() const;

  // Explicit ArrayView
  ArrayView<T> operator()() const;

  // Size and extent query functions
  size_type lowerOffset() const;
  size_type upperOffset() const;
  size_type size() const;

  // std::vector like and other misc functions
  void assign(size_type n, const T &val);
  template<class Iter>
    void assign(Iter first, Iter last);
  void deepCopy(const ArrayView<const T>& av);
  void resize(const size_type n, const T &val = T());
  void clear();

  // Common array class members (see above)
  ...

  // Reference counting (same as for RCP)
  ...

private:
  T *ptr_; // NULL if this pointer is null
  RCPNodeHandle node_; // NULL if this pointer is null
  size_type lowerOffset_; // 0 if this pointer is null
  size_type upperOffset_; // -1 if this pointer is null
};

// Nonmember constructors
```

```
template<class T>
ArrayRCP<T> arcp(T* p, typename ArrayRCP<T>::size_type lowerOffset,
  typename ArrayRCP<T>::size_type size, bool owns_mem = true);

template<class T, class Dealloc_T>
ArrayRCP<T> arcp(T* p, typename ArrayRCP<T>::size_type lowerOffset,
  typename ArrayRCP<T>::size_type size, Dealloc_T dealloc, bool owns_mem);

template<class T>
ArrayRCP<T> arcp( typename ArrayRCP<T>::size_type size );

template<class T>
ArrayRCP<T> arcpClone( const ArrayView<const T> &v );

template<class T>
ArrayRCP<T> arcp(const RCP<std::vector<T> > &v);

template<class T>
ArrayRCP<const T> arcp(const RCP<const std::vector<T> > &v);

template<class T>
ArrayRCP<T> arcpFromArrayView(const ArrayView<T> &av);

template<class T>
RCP<std::vector<T> > get_std_vector(const ArrayRCP<T> &ptr);

template<class T>
RCP<const std::vector<T> > get_std_vector(const ArrayRCP<const T> &ptr);

// Customized deallocators
...

// Embedded object functions
...

// Extra data functions
...

// Conversion functions
...

// Common non-member functions
...

// Other nonmember functions

template<class T>
typename ArrayRCP<T>::difference_type
operator-(const ArrayRCP<T> &p1, const ArrayRCP<T> &p2);

template<class T>
std::ostream& operator<<( std::ostream& out, const ArrayRCP<T>& p );
```

Some of the main features of the `ArrayRCP` class are:

`ArrayRCP` allows the user to allocate the contiguous array of data in anyway they would like and can define how that array is deallocated anyway they would like.

`ArrayRCP` returns persisting subviews of data through the member function `persistingView(...)`. This means that the underlying array of data will not be deleted until all the persisting subviews are destroyed.

`ArrayRCP` is a full replacement for a general raw pointer and can be used as a general iterator that always remembers the allowed upper and lower bounds. It supports all the appropriate pointer array-like operations including `ptr+i`, `i+ptr`, `ptr-i`, `ptr+=i`, `ptr-=i`, , `ptr++`, `ptr--`, `*ptr`, `ptr->member()`, and of course `ptr[i]`. This is what allows `ArrayRCP` to be used as a checked iterator implementation in a debug-mode build.

`ArrayRCP` can be used safely as a contiguous array by using it through its `const` interface which disables all of the pointer-like functions that change the frame of reference (e.g. `ptr+=i`, `ptr-=i`, , `ptr++`, and `ptr--` are disabled in the `const` interface).

`ArrayRCP` can be used in place of `std::vector` (and therefore `Array`) that only needs to size or resize the array in baulk and does not need to flexibly grow or shrink the array. It does this by supporting functions like `assign(...)`, `resize(...)`, and `clear()`. Because of the reference counting machinery that is always part of `ArrayRCP` and support for all raw C++ pointer functionality (e.g., `ptr++`), one may not want to use `ArrayRCP` instead of `Array` in many types of code. However, if the overhead is not going to be significant, then going with `ArrayRCP` instead of `Array` can be a good choice because it is much more flexible in how memory is allocated and has built-in support for shared ownership (again, which may not be needed). The class `ArrayRCP` does not attempt to replace `Array` but can be a better choice in many cases where an `Array` may otherwise be used.

`ArrayRCP` supports explicit shallow conversion to `ArrayView`. Requiring an explicit conversion from `ArrayRCP` to `ArrayView` in consistent with the required explicit conversion from `RCP` to `Ptr`. As explained in Section 5.8.4, requiring this type of explicit conversion is meant to increase the type safety and self-documenting nature of all code (including the calling code as well). Note that the `ArrayRCP::operator()()` function is a very short-hand way to perform conversion to `ArrayView`.

`ArrayRCP` supports owning conversions from `RCP`-wrapped `Array` and `std::vector` objects. This allows for better interoperability between code and uses solid reference-counting ownership semantics.

Some of the other features of the `ArrayRCP` class that are common with the other classes are discussed in Section 5.7, Section 5.9, and Section 5.11.

### 5.5.3 `Teuchos::Array<T>`

The class `Array` is a complete drop-in replacement for `std::vector` that is integrated with the `ArrayView` class for debug-mode runtime checking. In an optimized build, `Array` is nothing but an inline wrapper around a fully encapsulated `std::vector` object. This means that in an optimized build, `Array` takes advantage of all of the platform-specific optimizations contained in the native `std::vector` implementation and imparts no extra space/time overhead (see the timing results in Section 5.12.2 for evidence of this claim). However, in a debug build, a full set of platform-independent runtime checking is performed that is as strong or stronger than any checked STL implementation (see [31, Item 83]) and in

addition includes dangling reference detection of `ArrayView` views or direct conversions to `ArrayRCP` objects (see Section 5.11.3). `Array` also supports better runtime debug output with better exception error messages.

The class declaration for the `Array` class is shown in Listing 23.

**Listing 23** *: `Teuchos::Array` declaration (optimized build)*

```
template<typename T>
class Array {
public:

  // Constructors/initializers
  Array();
  explicit Array(size_type n, const value_type& value = value_type());
  Array(const Array<T>& x);
  template<ypename InputIterator> Array(InputIterator first, InputIterator last);
  Array(const ArrayView<const T>& a);
  template<int N> Array(const Tuple<T,N>& t);
  ~Array();
  Array& operator=(const Array<T>& a);

  // Other std::vector functions
  void assign(size_type n, const value_type& val);
  template<typename InputIterator> void assign(InputIterator first,
    InputIterator last);
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  size_type size() const;
  size_type max_size() const;
  void resize(size_type new_size, const value_type& x = value_type());
  size_type capacity() const;
  bool empty() const;
  void reserve(size_type n);
  reference operator[](size_type i);
  const_reference operator[](size_type i) const;
  reference at(size_type i);
  const_reference at(size_type i) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  void push_back(const value_type& x);
  void pop_back();
  iterator insert(iterator position, const value_type& x);
  void insert(iterator position, size_type n, const value_type& x);
  template<typename InputIterator> void insert(iterator position,
```

```
    InputIterator first, InputIterator last);
  iterator erase(iterator position);
  iterator erase(iterator first, iterator last);
  void swap(Array& x);
  void clear();

  // Conversions to and from std::vector
  Array( const std::vector<T> &v );
  std::vector<T> toVector() const;
  Array& operator=( const std::vector<T> &v );

  // Implicit conversion to ArrayView
  operator ArrayView<T>();
  operator ArrayView<const T>() const;

  // Common array class members (see above)
  ...

private:
  std::vector<T> vec_; // Optimized implementation
};


// Non-member helper functions
template<class T> ArrayRCP<T> arcp( const RCP<Array<T> > &v );
template<class T> ArrayRCP<const T> arcp( const RCP<const Array<T> > &v );
template<class T> ArrayRCP<T> arcpFromArray( Array<T> &a );
template<class T> ArrayRCP<const T> arcpFromArray( const Array<T> &a );
template<typename T> std::ostream& operator<<(std::ostream& os,
  const Array<T>& array);
template<typename T> std::vector<T> createVector( const Array<T> &a );
std::string toString(const Array<T>& array);
template<typename T> Array<T> fromStringToArray(const std::string& arrayStr);

// Other common nonmember functions
...
```

The usage of the `Array` class is identical to the usage of `std::vector` except that it naively supports the creation of `AraryView` objects that can detect and report dangling references or attempts to resize the container when one or more `ArrayView` objects are active. The unit tests for `Array` provide a complete catalog of all the debug-mode runtime checking that `Array` performs. A more general discussion of debug-mode runtime checking can be found in Section 5.11.

### 5.5.4 `Teuchos::Tuple<T,N>`

The last array class discussed here is the `Tuple` class which represents a compile-time sized array that implicitly converts into an `ArrayView` object. The class listing for `Tuple` is shown in Listing 24.

**Listing 24** *: `Teuchos::Tuple` declaration (optimized build)*

```
template<typename T, int N>
```

```
class Tuple {
public:

  // Constructors/initializers
  inline Tuple();
  Tuple( const Tuple<T,N> &t );

  // Implicit conversion to ArrayView
  operator ArrayView<T>();
  operator ArrayView<const T>() const;

  // Common array class members (see above)
  ...

private:
  T array_[N]; // Optimized implementation
};

// Non-member constructors

template<typename T>
Tuple<T,1> tuple(const T& a);

template<typename T>
Tuple<T,2> tuple(const T& a, const T& b);

template<typename T>
Tuple<T,3> tuple(const T& a, const T& b, const T& c);

...

template<typename T>
Tuple<T,15> tuple(const T& a, const T& b, const T& c, const T& d, const T& e,
  const T& f, const T& g, const T& h, const T& i, const T& j, const T& k,
  const T& l, const T& m, const T& n, const T& o);
```

The class `Tuple` is very small and efficient in an optimized build. All the functions are inlined and all data is allocated on the stack (or statically) and does not use the free store. In an debug build, however, `Tuple` takes on all the debug checking of all the other Teuchos array classes including the detection of dangling `ArrayView` views and dangling iterators.

One of the most useful features of `Tuple` is that a number of overloaded non-member constructor functions with name `tuple(...)` are provided (show above) to make it easy to pass in arrays to functions that accept them as `ArrayView` arguments. Overloads of `tuple(...)` are currently provided from one up through 15 arguments. For an example for using `tuple(...)` to call a function call, consider the function to be called:

```
void doSomething(const ArrayView<const int>&);
```

To call the function with three int arguments, one would use:

```
doSomething(tuple<int>(1, 2, 3)); // Implicitly converts to ArrayView<int>
```

Note that in an optimized build for the above function call that all data would be allocated on the stack and would not involve the free store. This results in very efficient code which is important when this is being used in an inner loop.

### 5.5.5 Array views

One of the most powerful features of the Teuchos memory management array types is that they allow for the creation of arbitrary contiguous subviews of data that have the strongest debug-mode runtime checking possible. All of the array classes `ArrayView`, `ArrayRCP`, `Array`, and `Tuple` provide contiguous views as `ArrayView` objects. The functions that provide `ArrayView` views are shown in Table 5. The `ArrayRCP` class can also provide persisting contiguous subviews as new `ArrayRCP` objects using the function `ArrayRCP::persistingView(...)`. Persisting views will remain even if the parent `ArrayRCP` objects have been released.

As soon as a contiguous array of data is correctly captured in `Array` or an owning `ArrayRCP` object, all children `ArrayView` objects will be protected in that if the parent array gets deleted, a debug-mode runtime check will detect and report a dangling reference if a client tries to access the data after the parent has gone away (see Section 5.11.3 for details).

To demonstrate the elegance and superior error checking of `ArrayView` subviews, consider a refactored version of code in Listing 4 and Listing 5 that tried to use `std::vector` but resulted in verbose clumsy code that was really no more correct or safe than the raw C++ pointer version. This refactored version to use `ArrayView` is shown in Listing 25 and Listing 26.

**Listing 25** *: Refactored version of Listing 4 to use `ArrayView`*

```
template<class T>
class BlockTransformerBase {
public:
  virtual ~BlockTransformerBase();
  virtual void transform(const ArrayView<const T> &a, const ArrayView<T> &b) const = 0;
};

template<class T>
class AddIntoTransformer : public BlockTransformerBase<T> {
public:
  virtual void transform(const ArrayView<const T> &a, const ArrayView<T> &b) const
    {
      DEBUG_MODE_ASSERT_EQUALITY( a.size(), b.size() );
      for (int i = 0; i < a.size(); ++i)
        b[i] += a[i];
    }
};
```

**Listing 26** *: Refactored version of Listing 5 to use `ArrayView`*

```
void someBlockAlgo( const BlockTransformerBase &transfomer,
  const int numBlocks, const ArrayView<const double> &big_a,
  const ArrayView<double> &big_b )
```

```
{
  DEBUG_MODE_ASSERT_EQUALITY( big_a.size(), big_b.size() );
  const int totalLen = big_a.size();
  const int blockSize = totalLen/numBlocks; // Assume no remainder!

  const int blockOffset = 0;
  for (int block_k = 0; block_k < numBlocks; ++block_k, blockOffset += blockSize)
  {
    if (big_a[blockOffset] > 0.0) {
      transformer.transform(big_a(blockOffset, blockSize),
        big_b(blockOffset, blockSize));
    }
  }
}
```

The advantages of the refactored code in Listing 25 and Listing 26 are that they are nearly as compact as the raw pointer versions in Listing 3 and Listing 6 but in addition also have full debug-mode runtime error checking. To see the improved safety, let's consider the case where the `transform(...)` function is incorrectly implemented with an off-by-one error as shown in Listing 27.

**Listing 27** *: Refactored version of off-by-one error in Listing 7 to use `ArrayView`*

```
template<class T>
void AddIntoTransformer<T>::transform(const ArrayView<const T> &a, const ArrayView<T> &b)
{
  DEBUG_MODE_ASSERT_EQUALITY( a.size(), b.size() );
  for (int i = 0; i <= a.size(); ++i)
    b[i] += a[i]; // Throws when i == a.size()
}
```

If the erroneous `transform(...)` function in Listing 27 were called from Listing 26 then in debug-mode, a runtime exception would immediately be raised when the `transform(...)` function tried to access one past the last element. As mentioned in Section 3.2, memory checking tools like Valgrind or Purify will never be able to catch semantic usage errors like this but it is trivial to catch these mistakes when using the Teuchos memory management classes.

As mentioned in Section 5.7, subviews can also be used along with the reinterpret cast functions to create very efficient memory management schemes for POD (plain old data) where large untyped `char` arrays are created and then subviews are broken off and reinterpret cast to specific data types. Examples of this can be found in the unit testing code.

## 5.6 Const versus non-const pointers and objects

The core smart-pointer pointer classes `Ptr`, `RCP`, `ArrayView` and `ArrayRCP` allow for the inner object (or array of objects) to be const or non-const and for the outer pointer object to be const or non-const, just like with regular C++ pointers. To draw the analogy with raw pointers, consider the equivalent declarations of a raw pointer and the pointer encapsulation class in Table 6.

**Equivalencies for const protection for raw pointers and Teuchos smart pointers types**

| Description | Raw pointer | Smart pointer |
|---|---|---|
| Basic declaration (non-const obj) | `typedef A* ptr_A` | `RCP<A>` |
| Basic declaration (const obj) | `typedef const A* ptr_const_A` | `RCP<const A>` |
| non-const pointer, non-const object | `ptr_A` | `RCP<A>` |
| const pointer, non-const object | `const ptr_A` | `const RCP<A>` |
| non-const pointer, const object | `ptr_const_A` | `RCP<const A>` |
| const pointer, const object | `const ptr_const_A` | `const RCP<const A>` |

**Table 6.** Equivalences between raw pointer and smart pointer types for const protection. Here, `RCP` is a stand-in for all four types `Ptr`, `RCP`, `ArrayView` and `ArrayRCP`.

The majority of problems that beginners have with the Teuchos memory management classes is related to the inability to make the basic equivalencies between raw pointers and smart pointers shown in Table 6 (see Section 5.7.3 for specific examples). It is critical that the programmer recognize this equivalence with raw pointers because it impacts many things especially implicit type conversions to satisfy function calls (again, see Section 5.7).

## 5.7 Conversions

Type conversions exist both for a single smart pointer type for the embedded type argument (e.g. `RCP<Derived>` implicitly converts to `RCP<const Derived>`, `RCP<Base>`, and `RCP<const Base>`) and also between different smart pointer types (e.g. `Array` implicitly converts to `ArrayView`). There are implicit conversions and explicit conversions. These two types of conversions are depicted in Figures 2 and 3 and shown in more detail in more detail in Table 8 and Table 9. (Note: All of the conversions shown in Table 8 and Table 9 are not shown in Figure 2 and Figure 3 for the sake of not making the figures to complex.) These conversions are described in the following two sections.

### 5.7.1 Implicit and explicit raw-pointer-like conversions

The core Teuchos memory management smart pointer types `Ptr`, `RCP`, `ArrayView` and `ArrayRCP` support all of the reasonable implicit and explicit type conversions that are defined by raw C++ pointers. C++ defines implicit conversions for raw pointers from non-const to const and from derived to base types. Table 7 shows what implicit and explicit conversions are supported for the four core memory management smart pointer types.

As seen in Table 7, the smart pointer types for single objects `Ptr` and `RCP` do not support the same implicit and explicit conversions that are supported for the array smart pointer types `ArrayView` and `ArrayRCP`. As explained in Section 2.2, it almost always incorrect and dangerous to allow implicit conversions from derived to base type for pointers that point into contiguous arrays of objects. Therefore, the types `ArrayView` and `ArrayRCP` do not support implicit conversions from derived to base types. Due to similar logic, it almost never makes any sense to perform a static cast or a dynamic cast on a pointer to an array of

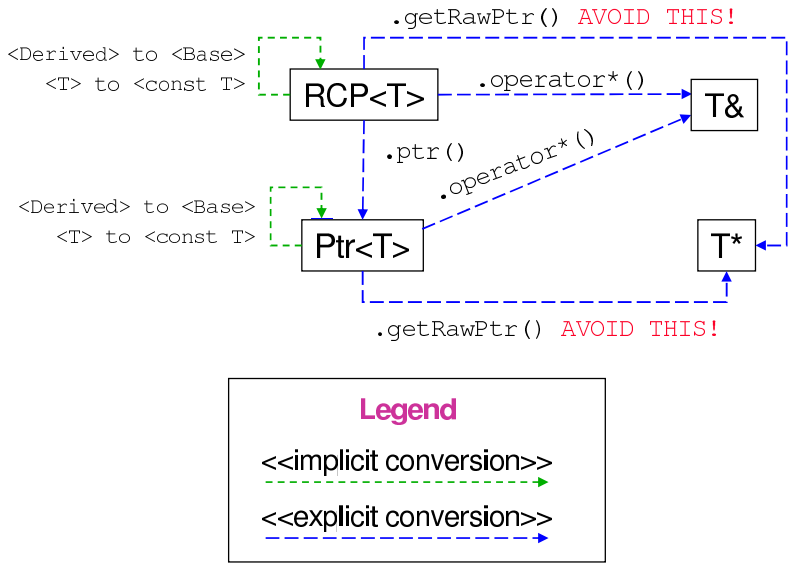



**Figure 2.** Conversions between different single-object memory management types.

**Basic implicit and explicit supported conversions for Teuchos smart pointer types**

| Operation | Ptr<T> | RCP<T> | ArrayView<T> | ArrayRCP<T> |
|---|---|---|---|---|
| Implicit conv derived to base | x | x | | |
| Implicit conv non-const to const | x | x | x | x |
| const_cast | x | x | x | x |
| static_cast | x | x | | |
| dynamic_cast | x | x | | |
| reinterpret_cast | | | x | x |

**Table 7.** Basic implicit and explicit conversions by smart-pointer types.

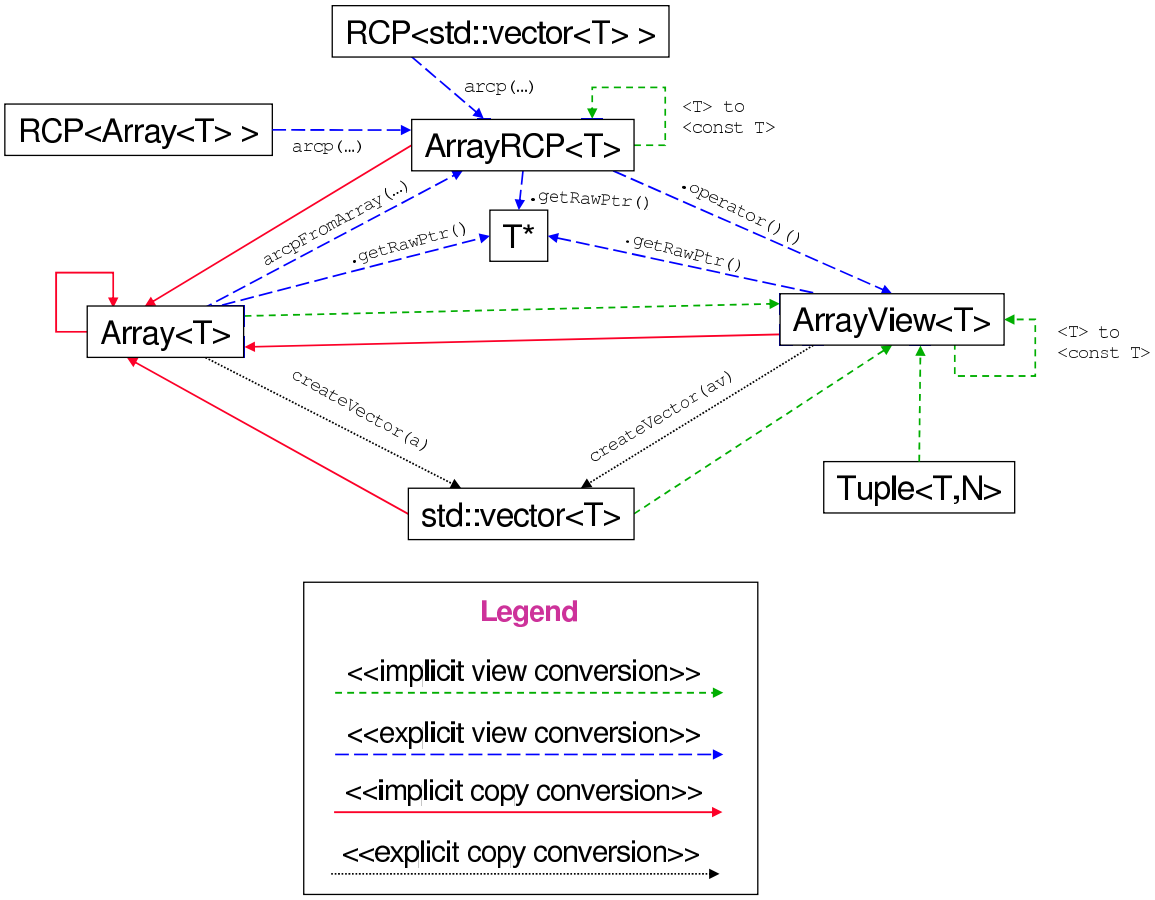



**Figure 3.** Conversions between array memory management types.

contiguous objects so the types `ArrayView` and `ArrayRCP` do not support static and dynamic casts.

In well formed programs, there is a justification to perform reinterpret casts for contiguous arrays of POD (plain old data) types. It is perfectly reasonable to allocate a large array of `char` (untyped) data and then create subviews and reinterpret cast to separate arrays of `double` and `int` data, for instance. However, in a well formed program in high-level code there is not a single valid reason to perform a reinterpret cast for single objects and therefore reinterpret cast is not supported for the types `Ptr` and `RCP`.

The allowed implicit conversions for `Ptr` and `RCP` are implemented through templated copy constructors (see Section 5.4.1 and Section 5.4.2). However, the only allowed implicit conversion for `<T*>` to `<const T*>` for `ArrayView` and `ArrayRCP` are instead supported through conversion member functions (see Section 5.5.1 and Section 5.5.2). The supported explicit conversion operators for these four types are shown in the Listings 28, 29, 30, and 31.

**Listing 28** *: Conversion functions for* `Ptr`

```
template<class T2, class T1> Ptr<T2> ptr_implicit_cast(const Ptr<T1>& p1);
template<class T2, class T1> Ptr<T2> ptr_static_cast(const Ptr<T1>& p1);
template<class T2, class T1> Ptr<T2> ptr_const_cast(const Ptr<T1>& p1);
template<class T2, class T1> Ptr<T2> ptr_dynamic_cast(const Ptr<T1>& p1,
  bool throw_on_fail = false);
```

**Listing 29** *: Conversion functions for RCP*

```
template<class T2, class T1> RCP<T2> rcp_implicit_cast(const RCP<T1>& p1);
template<class T2, class T1> RCP<T2> rcp_static_cast(const RCP<T1>& p1);
template<class T2, class T1> RCP<T2> rcp_const_cast(const RCP<T1>& p1);
template<class T2, class T1> RCP<T2> rcp_dynamic_cast(const RCP<T1>& p1,
  bool throw_on_fail = false);
```

**Listing 30** *: Conversion functions for ArrayView*

```
template<class T2, class T1> ArrayView<T2> av_const_cast(const ArrayView<T1>& p1);
template<class T2, class T1> ArrayView<T2> av_reinterpret_cast(const ArrayView<T1>& p1);
```

**Listing 31** *: Conversion functions for ArrayRCP*

```
template<class T2, class T1> ArrayRCP<T2> arcp_const_cast(const ArrayRCP<T1>& p1);
template<class T2, class T1> ArrayRCP<T2> arcp_reinterpret_cast(const ArrayRCP<T1>& p1);
template<class T2, class T1> ArrayRCP<T2> arcp_reinterpret_cast_nonpod(
  const ArrayRCP<T1>& p1, const T2& val = T2());
```

These conversion functions are used very similarly as for the built-in conversion operations in that only the output type needs to be explicitly specified. For example, Listing 32 shows some example conversions involving `RCP` (but the conversion function usage for the other types are identical). One function worth noting in Listing 31 is `arcp_reinterpret_cast_nonpod(...)` which performs a reinterpret cast from a POD (plain old data) datatype (e.g. `char`) to a non-POD datatype (e.g. `std::vector<int>`). This function calls (copy) constructors on the array elements and defines a specialized deallocation policy to call destructors on the elements when the last `ArrayRCP<T2>` object is released.

**Listing 32** *: Example usage of the explicit conversion functions*

```
RCP<const Base> = cbase(new Derived);
RCP<Base> base = rcp_const_cast<Base>(cbase);
RCP<const Derived> cderived = rcp_dynamic_cast<const Derived>(cbase, true);
RCP<const Derived> cderived2 = rcp_static_cast<const Derived>(cbase);
// NOTE: Static casting of Base to Derived is not safe when
// using virtual base classes or multiple inheritance.  Only dynamic
// casting is always safe with polymophpic types.
```

Note that the dynamic cast conversion functions `ptr_dynamic_cast()` and `rcp_dynamic_cast()` both take an option extra argument `throw_on_fail` that if set to `true` will result in an exception being thrown on a dynamic cast failure which is embedded with a very helpful error message (accessed through the `std::exception::what()` function).

### 5.7.2 Conversions between different memory management types

It is critical that all conversions between the various Teuchos memory management classes be performed using conversion code provided by the memory management classes or by associated helper functions in the Teuchos library. Client code should never convert between memory management types by exposing a raw C++ pointer. As soon as a raw C++ pointer is exposed, nearly all of the debug-mode runtime checking will be disabled. If a raw C++ pointer is exposed in order to perform a needed valid conversion, then either the programmer overlooked an already provided conversion function or the function needs to be added to Teuchos (please contact the developers of Teuchos at `teuchos-users@software.sandia.gov`).

Figures 2 and 3 show many of the types of conversions that are supported between the different memory management types. Specific conversions are shown in more detail in Tables 8 and 9. (Note: All of the conversions shown in Tables 8 and 9 are not shown in Figures 2 and 3 for the sake of not making the figures to complex.) For single objects, the conversions between different `RCP` and `Ptr` objects of various kinds shown in Figure 2 and Table 8 include both implicit and explicit conversions (but do not show the explicit conversion functions already shown in Listings 28 and 29). Conversions between different array types shown in Figure 3 and Table 9 include both implicit and explicit conversions and view and copy conversions yielding various types of conversions (but do not show the explicit conversion functions already shown in Listings 30 and 31).

The conversions shown in Tables 8 and 9 (and also in Listings 28, 29, 30, and 31) are the most basic conversions supported by the Teuchos memory management types but are not the only supported conversions. The see the full set of type conversions supported, consult the Doxygen generated documentation[8]. Note that full debug-mode runtime checking is fully enabled for every conversion between Teuchos memory management types, including full dangling-reference detection and reporting when creating non-reference-counting types `Ptr` and `ArrayView`. In general, dangling references cannot be detected when converting from raw C++ pointers `T*` and raw C++ references `T&` or for shallow views involving `std::vector`. However, there are a few special cases where non-owning `Ptr`, `RCP`, and `ArrayView`, `ArrayRCP` objects created from raw C++ pointers (or references) will be able to detect dangling references through the sophisticated debug-mode node tracing system (see Sections 5.11.3 and 5.11.6 for details).

[8] `http://trilinos.sandia.gov/packages/teuchos`

**Most Common Basic Conversions for Single Object Types**

| Type To | Type From | Properties | C++ code |
|---|---|---|---|
| RCP<A> | A* | Ex, Ow | rcp(a_p) [1] |
| RCP<A> | A* | Ex, NOw | rcp(a_p,false) [2] |
| RCP<A> | A& | Ex, NOw | rcpFromRef(a) |
| RCP<A> | A& | Ex, NOw | rcpFromUndefRef(a) |
| RCP<A> | Ptr<A> | Ex, NOw, DR | rcpFromPtr(a) |
| RCP<A> | boost::shared_ptr<A> | Ex, Ow, DR | rcp(a_sp) |
| RCP<const A> | RCP<A> | Im, Ow, DR | RCP<const A>(a_rcp) |
| RCP<Base> | RCP<Derived> | Im, Ow, DR | RCP<Base>(derived_rcp) |
| RCP<const Base> | RCP<Derived> | Im, Ow, DR | RCP<const Base>(derived_rcp) |
| boost::shared_ptr<A> | RCP<A> | Ex, Ow, DR | shared_pointer(a_rcp) |
| A* | RCP<A> | Ex, NOw | a_rcp.getRawPtr() [3] |
| A& | RCP<A> | Ex, NOw | *a_rcp [4] |
| Ptr<A> | A* | Ex, NOw | ptr(a_p) [2] |
| Ptr<A> | A& | Ex, NOw | outArg(a) [5] |
| Ptr<A> | RCP<A> | Ex, NOw, DR | a_rcp.ptr() |
| Ptr<A> | RCP<A> | Ex, NOw, DR | a_rcp() |
| Ptr<A> | RCP<A> | Ex, NOw, DR | ptrFromRCP(a_rcp) |
| Ptr<const A> | Ptr<A> | Im, NOw, DR | Ptr<const A>(a_ptr) |
| Ptr<Base> | Ptr<Derived> | Im, NOw, DR | Ptr<Base>(derived_ptr) |
| Ptr<const Base> | Ptr<Derived> | Im, NOw, DR | Ptr<const Base>(derived_ptr) |
| A* | Ptr<A> | Ex, NOw | a_ptr.getRawPtr() [3] |
| A& | Ptr<A> | Ex, NOw | *a_ptr() [4] |
| A* | A& | Ex, NOw | &a [3] |
| A& | A* | Ex, NOw | *a_p [3] |

Types/identifiers: `A* a_p`; `A& a`; `Ptr<A> a_ptr`; `RCP<A> a_rcp`; `boost::shared_ptr<A> a_sp`;

Properties: Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning, NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection (NOTE: All conversions are shallow conversions, i.e. copies pointers not objects.)

1. Constructing an owning `RCP` from a raw C++ pointer is strictly necessary but must be done with great care according to the commandments in Appendix B.
2. Constructing a non-owning `RCP` or `Ptr` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer and raw pointer manipulation should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section 5.2).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.
5. See other helper constructors for passing `Ptr` described in Section 5.4.1.

**Table 8.** Summary of basic conversions supported involving single objects.

**Most Common Basic Conversions for Contiguous Array Types**

| Type To | Type From | Properties | C++ code (or impl function) |
|---|---|---|---|
| ArrayRCP<S> | S* | Sh, Ex, Ow | arcp(s_p,0,n) [1] |
| ArrayRCP<S> | S* | Sh, Ex, NOw | arcp(s_p,0,n,false) [2] |
| ArrayRCP<S> | Array<S> | Sh, Ex, NOw, DR | arcpFromArray(s_a) |
| ArrayRCP<S> | ArrayView<S> | Sh, Ex, NOw, DR | arcpFromArrayView(s_av) |
| ArrayRCP<S> | ArrayView<S> | Dp, Ex, Ow | arcpClone(s_av) |
| ArrayRCP<S> | RCP<Array<S> > | Sh, Ex, Ow, DR | arcp(s_a_rcp) |
| ArrayRCP<const S> | RCP<const Array<S> > | Sh, Ex, Ow, DR | arcp(cs_a_rcp) |
| ArrayRCP<const S> | ArrayRCP<S> | Sh, Im, Ow, DR | ArrayRCP::operator()() |
| S* | ArrayRCP<S> | Sh, Ex, NOw | s_arcp.getRawPtr() [3] |
| S& | ArrayRCP<S> | Sh, Ex, NOw | s_arcp[i] [4] |
| ArrayView<S> | S* | Sh, Ex, NOw | arrayView(s_p,n) [1] |
| ArrayView<S> | Array<S> | Sh, Im, NOw, DR | Array::operator ArrayView() |
| ArrayView<S> | Tuple<S> | Sh, Im, NOw, DR | Tuple::operator ArrayView() |
| ArrayView<S> | std::vector<S> | Sh, Im, NOw | ArrayView<S>(s_v) |
| ArrayView<S> | ArrayRCP<S> | Sh, Ex, NOw, DR | ArrayRCP::operator()() |
| ArrayView<const S> | const Array<S> | Sh, Im, NOw, DR | Array::operator ArrayView() |
| ArrayView<const S> | const Tuple<S> | Sh, Im, NOw, DR | Tuple::operator ArrayView() |
| ArrayView<const S> | const std::vector<S> | Sh, Im, NOw | ArrayView(cs_v) |
| ArrayView<const S> | ArrayRCP<const S> | Sh, Ex, NOw, DR | ArrayRCP::operator ArrayView() |
| S* | ArrayView<S> | Ex, NOw | s_av.getRawPtr() [3] |
| S& | ArrayView<S> | Ex, NOw | s_av[i] [4] |
| Array<S> | S* | Dp, Ex | Array<S>(s_p,s_p+n) |
| Array<S> | std::vector<S> | Dp, Im | Array<S>(s_v) |
| Array<S> | ArrayView<S> | Dp, Im | Array<S>(s_av) |
| Array<S> | Tuple<S,N> | Dp, Im | Array<S>(s_t) |
| Array<S> | ArrayRCP<S> | Dp, Ex | Array<S>(s_arcp()); |
| std::vector<S> | Array<S> | Dp, Ex | s_a.toVector(); |
| S* | Array<S> | Ex, NOw | s_a.getRawPtr() [3] |
| S& | Array<S> | Ex, NOw | s_a[i] [4] |

Types/identifiers: `S* s_p; ArrayView<S> s_av; ArrayRCP<S> s_arcp; Array<S> s_a; Tuple<S,N> s_t; std::vector<S> s_v; RCP<Array<S> > s_a_rcp; RCP<const Array<S> > cs_a_rcp;`

Properties: Sh = Shallow copy, Dp = Deep copy (dangling references not an issue), Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning (dangling references not an issue), NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection for non-owning

1. It should never be necessary to convert from a raw pointer to an owning `ArrayRCP` object directly. Instead, use the non-member constructor `arcp<S>(n)`.
2. Constructing a non-owning `ArrayRCP` or `ArrayView` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section 5.2).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.

**Table 9.** Summary of basic conversions supported for contiguous arrays.

### 5.7.3 Implicit type conversion problems and shortcomings

Implicit conversions between different Teuchos memory management types, especially in templated application code, is one of the most confusing aspects of using these classes. As shown in Figures 2 and 3, many different implicit conversions are defined. An implicit conversion will only be performed by the C++ compiler to satisfy the formal arguments for a function call when several conditions are satisfied: a) when it is needed to call a function where no other better functions provide a better match, b) when only a single implicit conversion for each argument is sufficient, and c) when calling a non-template function (or a template function where all of the template arguments are explicitly specified). Also, the C++ compiler will not be able to do implicit conversions to satisfy a function call when ambiguous function calls exists. Explaining the behavior of these implicit conversions in C++ gets down to the low-level details of the C++ type system that many C++ programmers take for granted or don't understand all that well in the first place.

Almost all of the problems that programmers have with implicit conversions occur when trying to call functions where implicit conversions are required to satisfy the signature of the function. Some of these problems occur when developers fail to understand the C++ type system. Other problems are due to a fundamental handicap that smart pointer types have with respect to raw C++ pointers.

Implicit conversions of the Teuchos memory management classes (or any other C++ classes in any other library) needed to call a given function fail for one of the following reasons:

1. Implicit conversions to functions fail because the memory management types are not passed by const reference (or by value) and are mistakenly (or on purpose) passed by non-const reference. (This is a programming error.)
2. Implicit conversions fail because templated functions cannot perform implicit conversions in order to satisfy a call. (This is a language usability annoyance associated with templates but also represents a fundamental shortcoming of smart pointers compared to raw C++ pointers.)
3. Implicit conversions fail due to ambiguous overloaded calls to overloaded functions that would otherwise work just fine when raw C++ pointers are involved. (This is a fundamental shortcoming of smart pointers or any other class as compared to raw C++ pointers.)

Each of these types of problems are examined one at a time in the following three subsections.

#### Implicit conversions failing due to passing by non-const reference

First, consider implicit conversion problems caused by erroneously passing Teuchos memory management objects by non-const references instead of by const reference. Consider the user-written function in Listing 33 that mistakenly passes an `RCP` by non-const reference.

**Listing 33** *: User function with a bad pass by non-const reference problem*

```
class Base { ... };
class Derived : public Base { ... };
```

```
void someUserFunction(RCP<const Base> &base); // Should be 'const RCP<>&'

void someOtherUserFunction()
{
   RCP<Derived> derived(new Derived);
   someUserFunction(derived);  // Compile error, no implicit conversion!
   RCP<const Derived> cderived = derived;
   someUserFunction(cderived); // Compile error, no implicit conversion!
   RCP<Base> base = derived;
   someUserFunction(base);     // Complile error, no implicit conversion!
   RCP<const Base> cbase = base;
   someUserFunction(cbase);    // Compliles fine, exact match!
}
```

When user code tries to call `someUserFunction(...)` as shown in Listing 33, the C++ compiler refuses to perform the implicit type conversions because the compiler will never perform an implicit type conversion for an argument passed by non-const reference. This type of error is made at least once by most developers when they first start using the Teuchos memory management classes and they can't understand why the code does not compile. To understand why the implicit conversions in Listing 33 are not occurring, one must understand the C++ type system in how it handles basic type conversions. The C++ standard specifies that implicit type conversions to facilitate the call of a C++ function will only occur for arguments passed by value or by *const* reference. For example, a C++ compiler will convert an `int` into a `double` to call a function taking a `double` argument but only if the double is passed by value (i.e. `double x`) or by const reference (i.e. `const double& x`). The same holds true for C++ pointer types. Note that every pointer type (e.g. `int*`, `SomeType*`) is a new C++ value data type that is automatically defined by the compiler for every defined type. The C++ compiler also automatically defines implicit conversions between pointer types for `T*` to `const T*` and for `Derived*` to `Base*` (or combinations of both with `Derived*` to `const Base*`). While C++ pointer data types have a special place in the C++ language, they behave exactly like every other data type in C++ with respect to non-const references and implicit conversions. That is, if a pointer object is passed by non-const reference instead of by value, the compiler will refuse the perform the implicit conversion. For example, the equivalent code to Listing 33 replacing `RCP` with raw pointers shown in Listing 34 will also result in code that will not compile.

**Listing 34** *: User function with a bad pass by non-const reference problem using raw pointers*

```
typedef const Base* ptr_const_Base; // Equivalent to RCP<const Base>

void someUserFunction(ptr_const_Base &base); // Bad pass by non-const ref!

void someOtherUserFunction()
{
   Derived *derived = new Derived;
   someUserFunction(derived);   // Compile error, no implicit conversion!
   const Derived *cderived = derived;
   someUserFunction(cderived); // Compile error, no implicit conversion!
   Base *base = derived;
   someUserFunction(base);     // Complile error, no implicit conversion!
   const Base *cbase = base;
   someUserFunction(cbase);    // Compliles fine, exact match!
   delete derived;
```

```
}
```

The way to fix this problem is to pass the Teuchos memory management types (or any other type one wants the compiler to perform an implicit conversion on) by const reference. For example, fixing the code in Listing 33 to pass by const reference shown in Listing 35 results in code that compiles just fine with the C++ compiler performing all of the expected implicit conversions.

**Listing 35** *: User function with corrected pass by const reference*

```
void someUserFunction(const RCP<const Base> &base); // Now correct!

void someOtherUserFunction()
{
   RCP<Derived> derived(new Derived);
   someUserFunction(derived);  // Compiles fine, Derived* -> const Base*
   RCP<const Derived> cderived = derived;
   someUserFunction(cderived); // Compiles fine, const Derived* -> const Base*
   RCP<Base> base = derived;
   someUserFunction(base);     // Compiles fine, Base* -> const Base*
   RCP<const Base> cbase = base;
   someUserFunction(cbase);    // Compliles fine, exact match!
}
```

### Implicit conversions failing due to templated function

Another situation where implicit conversions will fail to be performed to satisfy a function call are when the function being called is a template function. The C++98 standard does not allow the implicit conversion of input arguments in order to call a template function [26, Item 45]. For example, consider the code in Listing 36 that fails to compile:

**Listing 36** *: Situation where implicit conversion fails due to a template function.*

```
template<class T> class Base { ... };
template<class T> class Derived : public Base<T> { ... };

template<class T>
void someTemplateUserFunction(const RCP<const Base<T> > &base);

template<class T>
void someOtherTemplateUserFunction()
{
   RCP<Derived<T> > derived(new Derived<T>);
   someTemplateUserFunction(derived);   // No implicit conv, no compmile!
   RCP<const Derived<T> > cderived = derived;
   someTemplateUserFunction(cderived);  // No implicit conv, no compmile!
   RCP<Base<T> > base = derived;
   someTemplateUserFunction(base);      // No implicit conv, no compmile!
   RCP<const Base<T> > cbase = base;
```

```
    someTemplateUserFunction(cbase);     // Exact match, compiles!
}
```

What is frustrating and yet interesting about this situation is that if the RCPs are replaced with raw pointers, as shown in Listing 37, the C++ compiler will perform the implicit type conversions just fine.

**Listing 37** *: Example where implicit conversion to call a template function works fine when using raw C++ pointers.*

```
template<class T>
void someTemplateUserFunction(const Base<T> *base);

template<class T>
void someOtherTemplateUserFunction()
{
  Derived<T> *derived = new Derived<T>;
  someTemplateUserFunction(derived);  // Okay, Derived<T>* -> const Base<T>*
  const Derived<T> *cderived = derived;
  someTemplateUserFunction(cderived); // Okay, const Derived<T>* -> const Base<T>*
  Base<T> *base = derived;
  someTemplateUserFunction(base);     // Okay, Base<T>* -> const Base<T>*
  const Base<T> *cbase = base;
  someTemplateUserFunction(cbase);    // Okay, exact match!
  delete derived;
}
```

Comparing the templated code in Listing 36 and Listing 37 it is clear that C++ assigns special privileges and abilities to the conversion of raw C++ pointer data types that are not afforded to any other data type. This is the first example of where smart pointer classes in C++ are put at a fundamental disadvantage with respect to raw C++ pointers. This is an unfortunate situation but the problem can be dealt with by either forcing the conversion of the input arguments or by explicitly specifying the template arguments as shown, for example, in Listing 38.

**Listing 38** *: Example of methods for addressing implicit conversions to allow the call of templated functions*

```
template<class T>
void someTemplateUserFunction(const RCP<const Base<T> > &base);

template<class T>
void someOtherUserTemplateFunction()
{
   RCP<Derived<T> > derived(new Derived<T>);
   // Force the conversion Derived<T>* -> const Base<T>*
   someTemplateUserFunction(RCP<const Base<T> >(derived));
   // or, specify template argument allowing implicit conversion
   // Derived<T>* -> const Base<T>*
   someTemplateUserFunction<T>(derived);
   ...
}
```

As shown in Listing 38, typically the least verbose way to call a template function that requires a conversion of input arguments is to just explicitly specify the template argument(s) which turns the template function into a regular function in the eyes of the C++ compiler and then implicit conversions will be allowed to satisfy the function call[9].

### Implicit conversions failing due to ambiguous overloaded function calls

The last situation to discuss where implicit conversions will fail to be performed for the Teuchos memory management types occurs when calling overloaded functions that require a conversion of the internal pointer type that would otherwise work just fine for raw C++ pointers. Consider the example code in Listing 39 showing the use of overloaded functions that differ in the const type of the object.

**Listing 39** *: Example of ambiguous calls to overloaded functions*

```
class Base { ... };
class Derived : public Base { ... };

void someUserFunction(const RCP<Base> &base);         // Overload #1
void someUserFunction(const RCP<const Base> &base);   // Overload #2

void someOtherUserFunction()
{
   RCP<Derived> derived(new Derived);
   someUserFunction(derived);  // Compile error, ambiguous call
   RCP<const Derived> cderived = derived;
   someUserFunction(cderived); // Compile error, ambiguous call
   RCP<Base> base = derived;
   someUserFunction(base);     // Okay, exact match for Overload #1
   RCP<const Base> cbase = base;
   someUserFunction(cbase);    // Okay, exact match for Overload #2
}
```

The reason that the first two function calls in Listing 39 with `RCP<Derived>` and `RCP<const Derived>` result in ambiguous function call compile errors is that the C++ compiler is not smart enough to know that a conversion from `RCP<Derived>` to `RCP<Base>` is better than a conversion from `RCP<Derived>` to `RCP<const Base>` which would allow the first function call to result in a call to Overload #1, for instance. However, if raw C++ pointers are used in same code, as shown in Listing 40, the compiler will make the right implicit conversions and call the right overloaded functions just fine.

**Listing 40** *: Example of implicit conversions for overloaded functions that work just fine for raw pointers*

```
void someUserFunction(Base *base);         // Overload #1
void someUserFunction(const Base *base);   // Overload #2

void someOtherUserFunction()
```

[9]Enabling emplicit conversions of input arguments for template functions with explicitly defined template arguments does not work on always work on even recent versions of the Sun C++ compiler.

```
{
   Derived *derived = new Derived;
   someUserFunction(derived);  // Calls Overload #1: Derived* -> Base*
   const Derived *cderived = derived;
   someUserFunction(cderived); // Calls Overload #2: const Derived* -> const Base*
   Base *base = derived;
   someUserFunction(base);     // Okay, exact match for Overload #1
   const Base *cbase = base;
   someUserFunction(cbase);    // Okay, exact match for Overload #2
   delete derived;
}
```

Again, similar to the templated function example given above, comparing Listing 39 and Listing 40, it is clear that the conversions of raw C++ pointer types to call overloaded functions are given special privileges and abilities that are not afforded to any other data type in C++. The C++ compiler will resolve overloaded functions for the conversion of C++ pointer types based on the least required conversions (e.g. `Derived*` to `Base*` is better than `Derived*` to `const Base*`). This is wonderful behavior for raw C++ pointers (or perhaps confusing depending on how one looks at it) but such special abilities are not afforded to smart pointer types like `Ptr` or `RCP` (or any other smart pointer type including `boost::shared_ptr`)[10].

Problems in calling overloaded functions like this can be resolved but only through explicitly converting the input arguments as shown in Listing 41.

**Listing 41** *: Example of resolving ambiguous calls to overloaded functions through explicit argument conversions*

```
void someUserFunction(const RCP<Base> &base);         // Overload #1
void someUserFunction(const RCP<const Base> &base);   // Overload #2

void someOtherUserFunction()
{
   RCP<Derived> derived(new Derived);
   someUserFunction(RCP<Base>(derived));        // Calls Overload #1
   someUserFunction(RCP<const Base>(derived));  // Calls Overload #2
   ...
}
```

Having to explicitly convert input arguments to satisfy overloaded function calls gets annoying very quickly for any reasonable-minded programmer. A much better way to deal with the problem of overload functions and smart pointer types is to not use overloaded functions in the first place as demonstrated in Listing 42.

**Listing 42** *: Example of resolving ambiguous calls to overloaded functions by not using overloaded functions in the first place*

```
void someNonconstUserFunction(const RCP<Base> &base);
void someUserFunction(const RCP<const Base> &base);
```

[10] Fixing the problem of implicit conversions for template and overloaded functions to put smart pointers at the same level as raw pointers would require a C++ language extension.

```
void someOtherUserFunction()
{
   RCP<Derived> derived(new Derived);
   // Compiles fine, implicit conv: Derived* -> Base*
   someNonconstUserFunction(derived);
   // Compiles fine, implicit conv: Derived* -> const Base*
   someUserFunction(derived);
   ...
}
```

Avoiding problems with ambiguous function calls to overloaded functions by avoiding overloaded functions (as demonstrated in Listing 42) may seem like a bit of cop-out but in general function overloading tends to be abused in C++ anyway. In many cases, code can be much more clear by using different function names in cases where most developers would just use overloaded functions (perhaps because they cannot think of better non-overloaded names).

## 5.8 Core idioms for the use of the Teuchos memory management classes

Well designed C++ class libraries are created together with a set of idioms for their use and this is especially true for the Teuchos Memory Management classes. This paper describes idioms related to the creation of single dynamically allocated objects, for defining and using local variables and data members, for passing objects and arrays of objects to and from functions, and for returning objects and arrays of objects as return values from functions. It is critical that these idioms be used consistently in order to yield the safest, highest quality, clearest, most self-documenting code.

### 5.8.1 The non-member constructor function idiom

The mainstream C++ literature espousing the use of smart reference-counted pointers like `boost::shared_ptr` seems to lack a solution for an effective, safe, and clean way to create new dynamically allocated objects. To demonstrate the issues involved, consider the C++ class `Blaget` shown in Listing 43:

**Listing 43** *: A class taking multiple dynamically allocatable objects*

```
class Blaget {
public:
  Blaget(const RCP<Widget> &widgetA, const RCP<Widget> const widgetB);
    widgetA_(widget), widgetB_(widget) {}
  ...
private:
  RCP<Widget> widgetA_;
  RCP<Widget> widgetB_;
};
```

Now consider how one might go about constructing a `Blaget` object on the stack given newly dynamically allocated `Widget` objects. A compact, clean, and seemingly safe way to do so is shown in Listing 44.

**Listing 44** *: A leaky way to construct*

```
Blaget blaget( rcp(new Widget()), rcp(new Widget()) );
```

The problem with the code in Listing 44 is that it might result in a memory leak if an exception is thrown by one of the constructors for `Widget` (see [31, Item 13]). The reason that a memory leak might occur is that a C++ compiler is allowed to evaluate both `new Widget()` calls before calling the `rcp()` functions. If the second constructor `Widget()` throws an exception after the first `Widget()` constructor has been invoked but before the `RCP` object wrapping the first `Widget` object is constructed, then the memory created by the first `new Widget()` will never be reclaimed.

The current C++ literature (see [31, Item 13]) recommends rewriting constructor code like shown in Listing 44 using temporary local variables as shown in Listing 45.

**Listing 45** *: A sound but verbose way to construct*

```
RCP<Widget> widgetA(new Widget());
RCP<Widget> widgetB(new Widget());
Blaget blaget(widgetA, widgetB);
```

While the code in Listing 45 will avoid a memory leak being created in case an exception is thrown, competent Java and Python programs will rightfully be disgusted that they have to create temporary variables just to call another constructor. From a software engineering perspective, it is undesirable to create useless local variables like `widgetA` and `widgetB` because they might be inadvertently copied and used for other purposes, resulting in undesirable side-effects.

The way to solve the problems described above is to provide non-member constructor functions for all dynamically allocatable reference-type classes and then always call them to create `RCP`-wrapped objects in client code. In fact, to avoid mistakes when using reference-type classes, one should disallow the creation of reference-type objects except through a provided non-member constructor. A *non-member constructor* compliant `Widget` class declaration is shown in Listing 46.

**Listing 46** *: The non-member constructor idiom for reference-type classes*

```
class Widget {
public:
  static RCP<Widget> create() { return rcp(new Widget); }
  void display(std::ostream&);
private: // or protected
  // Not for user's to call!
  Widget();
  Widget(const Widget&);
  Widget& operator=(const Widget&);
};

// Non-member constructor function
inline RCP<Widget> createWidget() { return Widget::create(); }
```

Using the non-member constructor function `createWidget()`, the unsafe constructor call in Listing 44 can be written as shown in Listing 47.

**Listing 47** *: Clean and bullet-proof way to construct dynamically allocated objects using the "non-member constructor function" idiom*

```
Blaget blaget(createWidget(), createWidget());
```

The code in Listing 47 will never result in a memory leak if an exception is thrown because each argument is returned as a fully formed `RCP` object which will clean up memory if any exception is thrown.

Note that the use of the *non-member constructor idiom* not only means that raw calls to `delete` are encapsulated in all high-level C++ code (due to the use of `RCP`), but it also means that raw calls to `new` should be largely encapsulated as well!

The non-member constructor idiom as shown in Listing 46 where a reference-type object can only be dynamically allocated and returned wrapped in an `RCP` object is recommended for all reference-type objects. The reason for this is that, as described in Section 5.9, when an object is dynamically allocated in managed in an `RCP` object, a number of important debug-mode runtime checks can be performed which cannot be when the object is first allocated on the stack or managed as a static object.

### 5.8.2 General idioms for handling arrays of objects

Before describing specific idioms for class data members, formal function arguments, and function return types it is worth discussing how arrays of objects are treated in a common way in all of these idioms and why. A common set of idioms that is used throughout is how arrays of value-type objects and reference-types objects are handled. When dealing with an array of value-type objects, typically a contiguous array of objects will be allocated. For example, to create an array of value-type objects one would declare:

```
Array<S> valTypeArray;
```

In this case, the storage for the array holding the value-type objects and the storage for the value-type objects themselves are one and the same. This is also true for persisting and non-persisting views of array of value-type objects represented as `ArrayRCP<[const] S>` and `ArrayView<[const] S>`, respectively. It is common for numerical programs to create very large arrays of value-type objects of integers and floating point numbers. Therefore, it is usually important to share these arrays and pass them around instead of creating copies. Because if this, it is typical to see `ArrayRCP<[const] S>` being used to share large value-type arrays of objects.

On the other hand, one cannot generally allocate a contiguous array of reference-type objects. Instead, one has to allocate and use a contiguous array of (smart) pointer objects that then point to individually allocated reference-type objects. For example, to store an array of dynamically allocated reference-type objects, one would declare:

```
Array<RCP<A> > refTypeArray;
```

Anyone familiar with object-oriented programming in C++ should already knows this, but they might be accustomed to allocating and working with arrays of raw pointers like `std::vector<T*>`. This is a really bad idea of course which is mentioned in Item 79 "Store only values and smart pointers in containers" in [31]. In this case, one can think of the storage for the array of `RCP` value-type objects and the storage for the reference-type objects of type `A` themselves to be different sets of storage. For example, one can change what `A` object is pointed to in the `RCP<A>` object stored in the contiguous array to without changing the `A` object itself such as in Listing 48:

**Listing 48** *: Code that changes memory in the contiguous array but does not touch the memory in the reference-type objects themselves*

```
void foo(Array<RCP<A> > &refTypeArray, const RCP<A> &someA)
{
  refTypeArray[0] = someA;
}
```

Note in Listing 48 that technically the memory stored in the array (of `RCP<A>` objects) was changed but the memory stored in the reference-type objects being pointed to where not changed at all. Likewise, one can change an `A` object itself without disturbing the array storage inside of the `Array<RCP<A> >` object itself such as shown in Listing 49:

**Listing 49** *: Code that changes the memory associated with the reference-type objects but does not change the memory of the contiguous array at all*

```
void foo(const Array<RCP<A> > &refTypeArray)
{
  refTypeArray[0]->someChange();
}
```

As opposed to arrays used to store value-type objects (e.g. `int`, `float`, `double`, `std::complex<double>`, etc.) which can be huge (with millions of elements) one typically does not create large arrays of reference-type objects. (Note that creating large arrays of reference-type objects would generally imply that the reference-type objects are small and cheap and therefore creating a large array of `RCP` objects could impart a large storage and runtime overhead as described in Section 5.12.1.) Since arrays of reference-type objects tend to be small in well designed programs, one usually does not care to share the array storage of `Ptr` or `RCP` objects itself, only the reference-type objects they point to. Because of this, one typically will not see `ArrayRCP<[const] RCP<[const] A> >` objects being passed around and stored. Instead, one would typically just pass `ArrayView<[const] RCP<[const] A> >` objects and then use this array to create a new `Array<[const] RCP<[const] A> >` object to copy the smart pointers. In general, we use arrays of `RCP` objects for representing persisting associations and arrays of `Ptr` objects for representing non-persisting associations when dealing with reference-type objects.

### 5.8.3 Idioms for class object data members and local variables

In general, class object data members and local variables represent a persisting relationship and therefore should have unique ownership or use reference counting. That means that the types `Ptr` and `ArrayView`

**Class Data Members for Value-Type Objects**

| Data member purpose | Data member declaration |
|---|---|
| non-shared, single, const object | `const S s_;` |
| non-shared, single, non-const object | `S s_;` |
| non-shared array of non-const objects | `Array<S> as_;` |
| shared array of non-const objects | `RCP<Array<S> > as_;` |
| non-shared statically sized array of non-const objects | `Tuple<S,N> as_;` |
| shared statically sized array of non-const objects | `RCP<Tuple<S,N> > as_;` |
| shared fixed-sized array of const objects | `ArrayRCP<const S> as_;` |
| shared fixed-sized array of non-const objects | `ArrayRCP<S> as_;` |

**Table 10.** Idioms for class data member declarations for value-type objects.

**Class Data Members for Reference-Type Objects**

| Data member purpose | Data member declaration |
|---|---|
| non-shared or shared, single, const object | `RCP<const A> a_;` |
| non-shared or shared, single, non-const object | `RCP<A> a_;` |
| non-shared array of const objects | `Array<RCP<const A> > aa_;` |
| non-shared array of non-const objects | `Array<RCP<A> > aa_;` |
| shared fixed-sized array of const objects | `ArrayRCP<RCP<const A> > aa_;` |
| "..." (const ptr) | `ArrayRCP<const RCP<const A> > aa_;` |
| shared fixed-sized array of non-const objects | `ArrayRCP<RCP<const A> > aa_;` |
| "..." (const ptr) | `ArrayRCP<const RCP<const A> > aa_;` |

**Table 11.** Idioms for class data member declarations for reference-types objects.

should almost never be used for class object data members or local variables (especially not for data members). However, local variables of type `Ptr` and `ArrayView` will be created in a function when that are created off other `Ptr` and `ArrayView` objects (passed through the formal argument list). The types `Ptr` and `ArrayView` will also be used as local variables when semi-persisting associations are involved (see Section 5.12.3 for an example).

Tables 10 and 11 give some idioms for class object data members. Usages for local variables are similar. Table 10 shows a few use cases involving value-type objects. Table 11 shows use cases involving reference-type objects. Every possible use case is not shown in these tables, only the most common ones. There is almost no end to the number of different types of data structures that can be created by embedding these memory management types in each other to address different needs. When creating these composite data structures one just needs to understand the implications for the selections of the class types and for the use of const.

It is important to note that an `RCP<S>` data member for a value-type object is not shown in Table 10. That is

because once one declares an `RCP` object pointing to a value-type object, at that point one is treating the value-type object with reference semantics so it would be considered to be a reference-type object (which takes one to Table 11). Again, most value-type class objects can be treated as reference-types in certain contexts (e.g. such as when dynamically allocating a large `Array` object so it can be shared and avoid expensive deep copy semantics).

Note that there are a few other important differences between the way that value-type objects and reference-type objects are handled. The main difference, obviously, is that one can hold a value-type object by value but not for a reference-type object. One can see this in how single objects are stored and how arrays of objects are declared in Table 11.

### 5.8.4 Idioms for the specification of formal arguments for C++ functions

Described here are idioms for the specification of the formal arguments for C++ functions that maximize compile-time and debug-mode run-time checking, yield near optimal raw pointer performance for non-debug-mode builds, and result in highly self-documenting code. A key component to this specification is that no raw C++ pointers are used. Raw pointers are the cause of almost all memory usage problems in C++. Raw C++ references, on the other hand, are safe to use as long as the object reference they are being used to point to is valid and no persisting association exists (see Section 5.4.3).

Tables 12 and 13 give conventions for passing single objects and arrays of objects for value-type and reference-type objects, respectively. In this specification, the Teuchos classes `Ptr`, `RCP`, `ArrayRCP`, and `ArrayView` are used as a means to pass objects of another type (shown as `S` and `A` in Tables 12 and 13). Conventions are shown for both passing in objects and for passing out objects through the formal arguments to C++ functions. Note that value-type objects can always be handled using reference semantics so all of the passing conventions in Table 13 apply equally as well for value-type objects as they do for reference-type objects. However, the conventions in Table 12 only apply to value-type objects that can be stored in contiguous arrays.

This specification addresses the five different properties that must be considered when passing an object to a function as a formal function argument (or passing back an object through a formal function argument):

- Is it a single object or an array of objects?
- Does the object or array of objects use value semantics or reference semantics?
- Is the object or array of objects changeable or non-changeable (i.e. const)?
- Is this establishing a persisting or non-persisting (or semi-persisting) association?
- Is the object or array of objects optional or required?

The first four of these properties are directly expressed in the C++ code in all cases shown in Tables 12 and 13. The specification for whether an argument or object is required or optional must be documented in the function's interface specification (i.e. in a Doxygen documentation `param` field). It is declared here that, by default, an argument passed through an `Ptr`, `RCP`, `ArrayView`, or `ArrayRCP` object will be assumed to be required (i.e. non-null) unless otherwise stated. The only exception for this implicit assumption for

**Passing IN Non-Persisting Associations to Value Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object (required) | `S s` **or** `const S s` **or** `const S &s` |
| single, non-changeable object (optional) | `const Ptr<const S> &s` |
| single, changeable object (required) | `const Ptr<S> &s` **or** `S &s` |
| single, changeable object (optional) | `const Ptr<S> &s` |
| array of non-changeable objects | `const ArrayView<const S> &as` |
| array of changeable objects | `const ArrayView<S> &as` |

**Passing IN Persisting Associations to Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const ArrayRCP<const S> &as` |
| array of changeable objects | `const ArrayRCP<S> &ss` |

**Passing OUT Persisting Associations for Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const Ptr<ArrayRCP<const S> > &as` |
| array of changeable objects | `const Ptr<ArrayRCP<S> > &as` |

**Passing OUT Semi-Persisting Associations for Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const Ptr<ArrayView<const S> > &as` |
| array of changeable objects | `const Ptr<ArrayView<S> > &as` |

**Table 12.** Idioms for passing value-type objects to C++ functions.

**Passing IN Non-Persisting Associations to Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object (required) | `const A &a` |
| single, non-changeable object (optional) | `const Ptr<const A> &a` |
| single, changeable object (required) | `const Ptr<A> &a` **or** `A &a` |
| single, changeable object (optional) | `const Ptr<A> &a` |
| array of non-changeable objects | `const ArrayView<const Ptr<const A> > &aa` |
| array of changeable objects | `const ArrayView<const Ptr<A> > &aa` |

**Passing IN Persisting Associations to Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const RCP<const A> &a` |
| single, changeable object | `const RCP<A> &a` |
| array of non-changeable objects | `const ArrayView<const RCP<const A> > &aa` |
| array of changeable objects | `const ArrayView<const RCP<A> > &aa` |

**Passing OUT Persisting Associations for Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const Ptr<RCP<const A> > &a` |
| single, changeable object | `const Ptr<RCP<A> > &a` |
| array of non-changeable objects | `const ArrayView<RCP<const A> > &aa` |
| array of changeable objects | `const ArrayView<RCP<A> > &aa` |

**Passing OUT Semi-Persisting Associations for Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const Ptr<Ptr<const A> > &a` |
| single, changeable object | `const Ptr<Ptr<A> > &a` |
| array of non-changeable objects | `const ArrayView<Ptr<const A> > &aa` |
| array of changeable objects | `const ArrayView<Ptr<A> > &aa` |

**Table 13.** Idioms for passing reference-type objects to C++ functions.

non-null objects is `const Ptr<const T>&` for single, non-changeable, non-persisting, objects where these always mean that the argument is optional. If such an argument is required, it is specified as `const T&`.

An array of value objects is passed as contiguous storage through an `ArrayView<S>` or `ArrayView<const S>` object. An array of reference objects, however, cannot be passed in contiguous storage for the objects themselves and instead must be passed as contiguous storage of (smart) pointers to the objects using `ArrayView<const Ptr<const A> >` for non-persisting associations or `ArrayView<const RCP<const A> >` for persisting associations. The `const` can be removed from the either `Ptr`/`RCP` or `A` depending on what is allowed to change or not change during the function call.

Note that in the case of `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` objects, that these can be treated as output objects in their own right which is shown in Tables 12 and 13 for passing out persisting and semi-persisting relationships to single objects and arrays of objects. For example, passing an `RCP<T>` object into a function to be set to point to a different `A` object would be specified in the function prototype as `const Ptr<RCP<A> >&` or `RCP<A>&` depending on preference (only the case `const Ptr<RCP<A> >&` is shown in the tables which is a better self-documenting form and provides better debug-mode runtime checking since it can detect dangling references). Note that semi-persisting associations are always passed out as `Ptr` and `ArrayView` objects. These types have essentially zero overhead in an optimized build but yet have full runtime checking including detection and reporting of dangling references in a debug-mode build (see Section 5.12.3 for a discussion of the motivation and usage of semi-persisting associations). The types `RCP` and `ArrayRCP` are always used to establish persisting associations.

#### Variations in passing single changeable objects

The only area of contention in this specification is how to handle arguments for required single changeable objects. The specification described here allows either passing them through a smart pointer as `const Ptr<T>&` or as a raw non-const object reference as `T&`. In Item 25 in [31], the authors recommend passing a raw non-const object reference `T&` for changeable required objects, which seems very reasonable. However, other notable authors [29, Section Section 5.5] and [24, Section 13.2] and the Google C++ coding standard[11] recommend passing a pointer instead, as it provides a visual clue that the object is being modified in the function call. Of course, the idioms defined here do not allow raw pointers so one must pass a `const Ptr<T>&` object instead. To consider the issues, for example, looking at the function call in Listing 50, which (if any) argument(s) are being modified?

**Listing 50** *: Function call using all raw references where it is impossible to determine what objects are modified in the call*

```
someFunction(a, b, c, d);
```

To tell for sure which objects are being modified in Listing 50, one would have to look at the function prototype shown in Listing 51 to see that it is the `d` argument that is being modified in the function call.

**Listing 51** *: Function prototype where all objects are passed as raw C++ references*

```
void someFunction(const A& a, const B& b, const C& c, D& d);
```

[11] `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

Now consider the convention that all changeable arguments be passed in through a pointer as `const Ptr<T>&`, giving the new prototype shown in Listing 52.

**Listing 52** *: Function prototype where modified objects are passed through `Ptr` leading to self-documenting client code*

```
void someFunction(const A& a, const B& b, const C& c,
  const Ptr<D>& d );
```

Now the new function call in Listing 53 is self-documenting with regards to which object is modified in the function call by using the `outArg(...)` templated non-member function (see `outArg(...)` in Listing 19).

**Listing 53** *: Self-documenting function call that shows what argument is modified in the function call*

```
someFunction(a, b, c, outArg(d));
```

Also, given that all `Ptr<T>` arguments are assumed to be non-null by default, this specifies that passing an argument as `const Ptr<T>&` has all of the same meaning that passing an argument by `T&`. Of course now one has given up a compile-time check for a non-null argument for `T&` with a debug-mode runtime check that `const Ptr<T>&` is non-null. Theoretically, the compile-time check would appear to be far superior but in reality the debug-mode runtime check is usually what happens anyway since the raw object reference would typically be created from a smart pointer in most cases (which can be null, resulting in a null dereference runtime exception in a debug-mode build). Therefore the issue is not whether a compile-time check will catch passing a null-object (because it can't) but instead the issue is how soon a debug-mode runtime check will catch a dereference of a null smart pointer.

### Converting from non-persisting to persisting references to satisfy the defined idioms

There are legitimate instances where client code needs to convert a non-persisting reference (i.e. `T&`, `Ptr<T>`, or `ArrayView<T>`) to a persisting reference (i.e. `RCP` or `ArrayRCP`) in order to satisfy the idioms outlined in Tables 12 and 13. The most common case is when a function is passed a raw reference or a `Ptr` to a C++ object (for a non-persisting association) but the function's implementation must create (and destroy) and object what has a persisting association with the passed in object. Consider the classes `A`, `B` and `C` shown in Listing 9 where `C` maintains an `RCP` to `B`. Now consider a client function that needs an `A` and `B` object to perform its function but also needs to create and destroy a `C` object internally giving it the `B` object. In order to be consistent with the idioms defined here, the `B` object must be passed as a raw C++ reference or through a `Ptr` object. Listing 54 shows how to convert from a raw C++ reference to a non-owning `RCP` object to satisfy the idioms.

**Listing 54** *: Converting from a raw C++ reference to an `RCP` object to satisfy function argument passing idiom*

```
void doSomeOperation(B &b, const A &a)
{
```

```
    ...
    C c;
    const RCP<B> b_rcp = rcpFromRef(b);
    c.fooC1(b_rcp, a);
    c.fooC2();
    ...
    // The C object is destroyed here!
}
```

In Listing 54, the standard conversion function `rcpFromRef(...)` converts from a raw C++ reference to a non-owning `RCP` object. Creating an `RCP` like is perfectly safe and correct. The lifetime of the created `C` object is contained within the function `doSomeOperation(...)` so the promise of not creating a persisting association inherent in the functions prototype (i.e. passing the `B` object as a raw C++ reference) is being correctly kept. Note that if the created non-owning `RCP` is accidentally used to create a persisting association then, in many cases, the dangling reference will be caught by the built-in debug-mode runtime checking (see Section 5.11.3).

A similar type of conversion is required when passing in an object through a `Ptr` object. For example, the function in Listing 54 may instead pass in a `Ptr<B>` object instead of a raw C++ reference `B&` and the refactored function is shown in Listing 55.

**Listing 55** *: Converting from a `Ptr` object to an `RCP` object to satisfy function argument passing idiom*

```
void doSomeOperation(const Ptr<B> &b, const A &a)
{
    ...
    C c;
    const RCP<B> b_rcp = rcpFromPtr(b);
    c.fooC1(b_rcp, a);
    c.fooC2();
    ...
    // The C object is destroyed here!
}
```

Again, if a persisting association is accidentally created by copying the `RCP<B>` object created in Listing 55 then this can be detected in a debug-mode build. Note that the conversion from `Ptr<B>` to `RCP<B>` shown in in Listing 55 actually generates much more efficient code in a debug-mode build because dangling-reference detection is implemented without having to perform a more expensive node look-up as described in Section 5.11.3.

### 5.8.5 Idioms for returning objects from C++ functions

Idioms for how objects are returned from C++ functions are also important in order to achieve C++ code that is efficient, safe (with both compile-time and debug-mode run-time checking), and is as self-documenting as possible. Tables 14 and 15 give common specifications for returning single objects and arrays of objects for both value-type and reference-type objects for non-persisting, persisting, and semi-persisting associations. Five different types of properties that must be defined and considered when returning an object (or array of objects) from a function:

**Returning Non-Persisting Associations to Value Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single copied object (return by value) | `S` |
| Single non-changeable object (required) | `const S&` |
| Single non-changeable object (optional) | `Ptr<const S>` |
| Single changeable object (required) | `S&` |
| Single changeable object (optional) | `Ptr<S>` |
| Array of non-changeable objects | `ArrayView<const S>` |
| Array of changeable objects | `ArrayView<S>` |

**Returning Persisting Associations to Value Objects**

(Use cases not covered by reference semantics used for value types)

| Purpose | Return Type Declaration |
|---|---|
| Array of non-changeable objects | `ArrayRCP<const S>` |
| Array of changeable objects | `ArrayRCP<S>` |

**Returning Semi-Persisting Associations to Value Objects**

(Use cases not covered by reference semantics used for value types)

| Purpose | Return Type Declaration |
|---|---|
| Array of non-changeable objects | `ArrayView<const S>` |
| Array of changeable objects | `ArrayView<S>` |

**Table 14.** Idioms for returning value-type objects from C++ functions.

- Is it a single object or an array of objects?
- Does the object or array of objects use value semantics or reference semantics?
- Is the object or array of objects changeable or non-changeable (i.e. const)?
- Is this establishing a persisting or non-persisting (or semi-persisting) association?
- Is the object or array of objects optional or required?

These five different properties are the same five described for formal function arguments described in Section 5.8.4. Again, the first four of these properties are clearly defined in the C++ code itself. However, again, it is not always possible to directly state in the C++ code declarations whether the object (or array of objects) is optional or required. Here, we state by default that all array arguments of type `ArrayView` and `ArrayRCP` are assumed to be required non-null arguments by default. Otherwise, documentation must exist stating that the arguments are optional.

The semantics of return objects is different than for formal function arguments. There are several differences that one can see from looking at Tables 12 and 13, and Tables 14 and 15. The key difference between formal functions arguments and return values relates to using constant references for formal arguments versus returning objects by value as return types. While the memory management objects of

**Returning Non-Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single cloned object | `RCP<A>` |
| Single non-changeable object (required) | `const A&` |
| Single non-changeable object (optional) | `Ptr<const A>` |
| Single changeable object (required) | `A&` |
| Single changeable object (optional) | `Ptr<A>` |
| Array of non-changeable objects | `ArrayView<const Ptr<const A> >` |
| Array of changeable objects | `ArrayView<const Ptr<A> >` |

**Returning Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single non-changeable object | `RCP<const A>` |
| Single changeable object | `RCP<A>` |
| Array of non-changeable objects | `ArrayView<const RCP<const A> >` |
| Array of changeable objects | `ArrayView<const RCP<A> >` |

**Returning Semi-Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single non-changeable object | `Ptr<const A>` |
| Single changeable object | `Ptr<A>` |
| Array of non-changeable objects | `ArrayView<const Ptr<const A> >` |
| Array of changeable objects | `ArrayView<const Ptr<A> >` |

**Table 15.** Idioms for returning reference-type objects from C++ functions.

type `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` are all passed by constant reference in Tables 12 and 13, alternatively they are always returned as objects (i.e. return by value) in Tables 14 and 15. The reason that these memory management objects should always be returned by value is that this is needed to correctly set up the reference counting machinery to properly set up persisting relationships and to enable debug runtime checking (e.g. to detect dangling references with semi-persisting associations).

Note that it is critical that semi-persisting associations for single objects must always be returned as `Ptr<T>` objects and never as raw references `T&` which is otherwise acceptable for non-persisting associations. The reason that `Ptr<T>` objects must always be used for semi-persisting associations is that in a debug-mode build, the runtime checking machinery will be able to detect dangling references or changes in the parent object that would otherwise invalidate the semi-persisting view that is impossible to catch when using raw C++ references.

In the following section, an extended example is given highlighting the need to return the Teuchos memory management smart pointer objects by value. If one already accepts the need for this, the example can be skipped.

#### Extended example for the need to return smart pointers by value

In order to understand the importance of returning memory management objects by value instead of by reference, first consider Listing 56 that looks to be perfectly safe code.

**Listing 56** *: A seemingly safe use of raw C++ references*

```
void seeminglySafeFoo(Blob &blob, Flab &flab)
{
  blob.doGoodStuff(flab);
}
```

The code in Listing 56 does not itself look unsafe. However, the reason that it unsafe comes from the code that calls `seeminglySafeFoo(...)` and the code that implements `Blob` shown in Listing 57.

**Listing 57** *: Code that makes seeminglySafeFoo(...) fail*

```
class Blob
{
  RCP<Flab> flab_;
public:
  Blob() : flab_(createFlab()) {}
  const RCP<Flab>& getFlab() { return flab_; }
  void doGoodStuff(Flab &flab_in)
  {
    flab_ = createFlab(); // Using non-member constructor
    flab_in.conflab(*flab_);
  }
};
```

```
void badCallingFunction()
{
  Blob blob;
  seeminglySafeFoo(blob, *blob.getFlab());
}
```

When the code in Listings 56 and 57 executes, it will most likely cause a segfault when it runs, if one is lucky. However, if unlucky, the code will actually seem to be working correctly on the machine where the code is initially tested it but will explode later (perhaps years later) when run under different circumstances. The reason that the code in Listings 56 and 57 is faulty is because the `Flab` object that is passed through the call `seeminglySafeFoo(blob, *blob.getFlab())` to `blob.doGoodStuff(flab)` is invalidated before it is used because it gets destroyed and is replaced by a new object in the expression `flab_ = createFlab()`. When this happens, the object now represented as the raw C++ reference `flab_in` is deleted which causes the code in the expression `flab_in->conflab(*flab_)` to be in error, and the behavior of the program is undefined (and again will segfault if one is lucky).

How did it come to this situation? What if the raw C++ references were replaced with with RCP-wrapped objects? Well, consider the updated code in Listing 58.

**Listing 58** *: Still unsafe code*

```
class Blob
{
  RCP<Flab> flab_;
public:
  Blob() : flab_(createFlab()) {}
  const RCP<Flab>& getFlab() { return flab_; }
  void doGoodStuff(const RCP<Flab> &flab_in)
  {
    flab_ = createFlab(); // Using non-member constructor
    flab_in.conflab(*flab_);
  }
};


void seeminglySafeFoo(Blob &blob, const RCP<Flab> &flab)
{
  blob.doGoodStuff(flab);
}


void badCallingFunction()
{
  Blob blob;
  seeminglySafeFoo(blob, blob.getFlab());
}
```

Is the code in Listing 58 correct? The sad answer is no, it is not. The `Flab` object returned from `blob.getFlab()` will still get deleted before it is used in the expression `flab_->conflab(flab_in)`.

What is going on here? The core of the problem is that the function `Blob::getFlab()` is incorrectly implemented. Functions must always return `RCP` objects by value and never by reference as shown in Tables 14 and 15. By returning a raw C++ reference to the `RCP<Flab>` object, a persisting association with the client is never properly established and this is the root cause of the whole problem.

Now consider the updated code in Listing 59 that goes back to using raw C++ references where appropriate but now returns the `RCP<Flab>` object by value as it should.

**Listing 59** *: Correctly returning RCP by value yielding safe code*

```
class Blob
{
  RCP<Flab> flab_;
public:
  Blob() : flab_(createFlab()) {}
  RCP<Flab> getFlab() { return flab_; } // Returns by value now!
  void doGoodStuff(Flab &flab_in)
  {
    flab_ = createFlab(); // Using non-member constructor
    flab_in.conflab(*flab_);
  }
};


void seeminglySafeFoo(Blob &blob, Flab &flab)
{
  blob.doGoodStuff(flab);
}


void goodCallingFunction()
{
  Blob blob;
  seeminglySafeFoo(blob, *blob.getFlab());
}
```

Is the code represented in Listing 59 now safe and correct? Yes it is. The reason that it is now safe and correct is that a persisting relationship is now being correctly created by the function call `blob.getFlab()` in that a new temporary `RCP<Flab>` object is created (which increments the reference count). From this new temporary `RCP<Flab>` object, a raw C++ reference is then returned from `*blob.getFlab()` and passed through. In this case, since the reference count on the existing `Flab` object is now two instead of one, the expression `flab_ = createFlab()` will not delete the existing `Flab` object and the following expression `flab_in.conflab(*flab_)` will have two valid `Flab` objects. After the function `seeminglySafeFoo(blob, *blob.getFlab())` exits, the first `Flab` object will finally be deleted (but that is just fine).

### More examples of function return issues

Another difference between formal function arguments and return values is what persisting and non-persisting associations mean related to function returns. In the case of objects returned from C++

functions, a persisting association is one where the object returned from a C++ function is remembered past the end of the statement where the C++ function returning the objects is called. For example, consider the code in Listing 60.

**Listing 60** *: Example of a bad persisting association implemented as a raw C++ reference (see the* `Grob` *class defined in Listing 61)*

```
void foo(Glob& glob)
{
  const Flab &flab = glob.getFlab();
  glob.doStuff();
  flab.doMoreStuff();
}
```

The code in Listing 60 represents a persisting association because because the `Flab` object returned in the expression `const Flab &flab = glob.getFlab()` is remembered past the statement where it is called and is used later in calling `flab.doMoreStuff()`. This type of code is all too common in C++ programs (including a lot of code I have written over the last 10 years) but it is not safe because it is not properly respecting the notion of persisting associations. To see why the code in Listing 60 is so bad, consider the possible unfortunate implementation of the `Glob` class shown in Listing 61:

**Listing 61** *: Bad implementation of the* `Glob` *class with respect to persisting associations*

```
class Glob {
  RCP<Flab> flab_;
public:
  Glob() : flab_(createFlab()) {}
  const Flab& getFlab() const { return *flab_; }
  void doStuff()
    {
      flab_ = createFlab(); // Non-member constructor
      ...
    }
};
```

What happens of course is that the behavior of the code in Listings 60 and 61 is undefined and will most likely result in a segfault (if one is lucky). The reason this is bad code is that the `Flab` object reference that gets returned from `glob.getFlab()` is not used until after the function `Glob::doStuff()` gets called which will delete the `Flab` object and replace it with another one. This results in `flab.doMoreStuff()` being called on a deleted object. Again, this will typically result in a segfault, but on some systems in some cases the program might actually seem to run just fine, perhaps even for years. This of course is an error that a tool like Valgrind or Purify would likely catch pretty easily which is why these tools are very useful to have around. So what rule was broken in Listing 60? Consider again the definition of a persisting association related to a return value which is:

- *Persisting associations* are associations that exist between two or more objects that extend past a single function call for formal function arguments, or a single statement for function return objects.

What this means is that any object that is returned as a raw C++ reference from a function must be used in the same statement from where the returning function is called. Therefore, the function in Listing 60 should be rewritten as shown in Listing 62.

**Listing 62** *:*

```
void foo(Glob& glob)
{
  glob.getFlab().doMoreStuff();
  glob.doStuff();
}
```

Here, of course, one is assuming that the order of evaluation of the functions is not important.

Note that functions returning raw C++ references are common and are fairly safe to use as long as the returned object is used in the same statement where the function is called. For example, this is what is commonly done when a non-const reference to an element from a user-defined array class object is returned and set in the same statements such as shown in Listing 63.

**Listing 63** *:*

```
void foo(std::vector<int>& a)
{
  a[0] = 5; // Non-persisting function return association
  ...
}
```

What is typically not safe, of course, is when one tries to save a reference to an object and then use it like in Listing 64.

**Listing 64** *:*

```
void foo(std::vector<int>& a)
{
  int &a_0 = a[0]; // Incorrect persisting association
  a.resize(20);
  a_0 = 5;         // Will likely segfault if one is lucky!
  ...
}
```

The problem with the code in Listing 64 is that the `a.resize(20)` function might cause a new buffer to be allocated and the existing buffer to be deleted. This will of course make the reference returned in `int &a_0 = a[0]` invalid when it is later written to in `a_0 = 5`.

The whole point of the example code Listings 63 and 64 is to demonstrate the working definition of persisting & non-persisting associations as they relate to objects returned from functions. This argument supports the idioms shown in Tables 14 and 15.

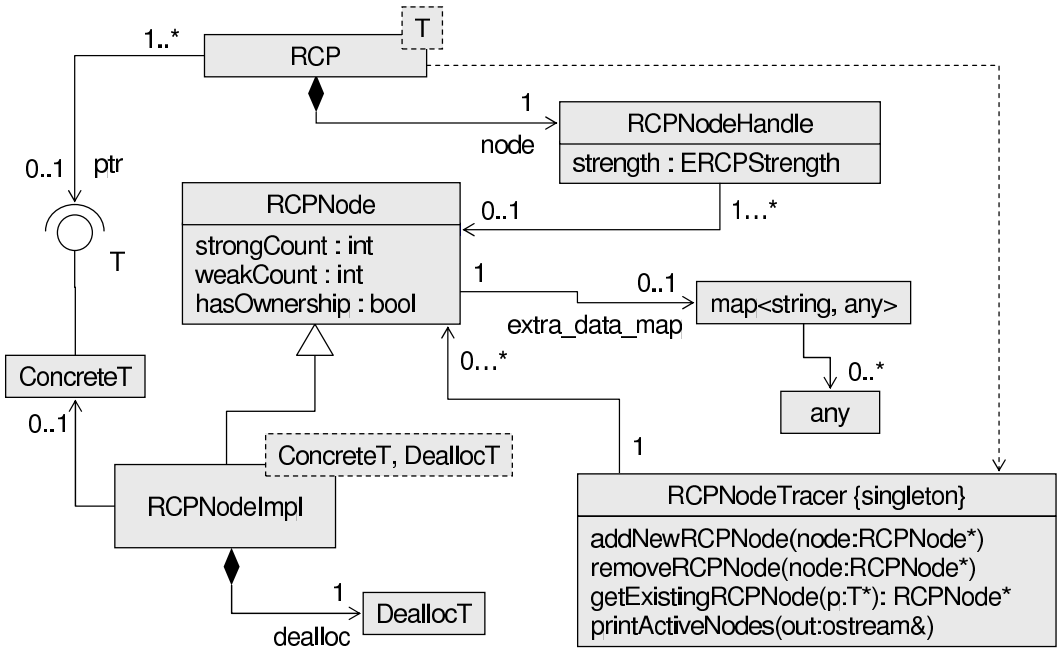



**Figure 4.** Basic design of the Teuchos reference-counting machinery.

## 5.9 Reference-counting machinery in-depth

In order to effectively use these memory management classes and to debug problems when they occur, one must understand the basic reference-counting approach being used. Basic reference counting with smart pointers is well established in the C++ literature [25] but a basic overview and specific details about the approach used in the Teuchos memory management classes is appropriate to describe here. Of equal importance is to describe how the reference-counting infrastructure can be used to address some boundary cases that can help solve some fundamental problems with reference counting.

The basic reference counting machinery being used by the classes is first described. Next, the issue of circular references and weak pointers are discussed.

### 5.9.1 Basic reference counting machinery

The foundation for the reference-counting machinery used by all of the reference-counting classes is shown in Figure 4 (UML class diagram). The class `RCPNode` is an abstract base class that contains two different reference counts (a strong count and a weak count) and inline functions for manipulating the counts as efficiently as possible. The templated concrete subclass `RCPNodeImpl` is what actually stores the raw C++ pointer to the reference-counted object. This class is also templated on an deallocation policy object that determines how the object is reclaimed. `RCPNodeHandle` is a simple handle class that automates the manipulation of the reference counts by overloading the copy constructor and assignment operator functions. This avoids having to replicate reference counting incrementing and decrementing in the

user-level classes RCP and ArrayRCP that contain it. All of the functions on RCPNodeHandle are inlined and the only data members are a pointer to the underlying RCPNode object and a strength attribute (with values RCP_STRONG and RCP_WEAK). The class RCPNodeHandle imparts zero space and time overhead and removes all duplication in how the reference count node object is handled. In future UML diagrams, the RCPNodeHandle class will be considered to be part of the owning RCP or ArrayRCP classes to avoid clutter. The classes RCPNode, RCPNodeImpl, and RCPNodeHandle, are used unchanged for both the RCP and ArrayRCP classes (however, only the RCP class is shown for simplicity).

The member functions for RCP and ArrayRCP related to reference-counting are shown in Listing 65.

**Listing 65** *: Reference counting member functions for RCP and ArrayRCP*

```
template<class T>
class [Array]RCP {
public:
  ...
  // Reference counting member functions
  ERCPStrength strength() const;
  bool is_valid_ptr() const;
  int strong_count() const;
  int weak_count() const;
  int total_count() const;
  void set_has_ownership();
  bool has_ownership() const;
  Ptr<T> release();
  RCP<T> create_weak() const;
  RCP<T> create_strong() const;
  template<class T2> bool shares_resource(const RCP<T2>& r_ptr) const;
  const RCP<T>& assert_not_null() const;
  const RCP<T>& assert_valid_ptr() const;
  ...
};
```

Most of the functions in Listing 65 are never called by general clients except in desperate situations. Notable exceptions are the member functions create_weak() (which is used to create a WEAK RCP object from a STRONG object) and and create_strong() (which is used to create a STRONG RCP object from a WEAK object). The function create_weak() is used to break a circular reference as described in Section 5.9.2 while create_strong() is used in situations like the "object self-reference" idiom described in Section 5.13.3.

Figure 4 also shows that every RCPNode object has an optional std::map object that can be used to store and retrieve arbitrary extra data (represented as the any data-type which can handle any value-type object). A raw pointer is stored to the extra_data_map object that is initialized to null by default. Therefore, if no extra data is used, the only overhead for this feature is an extra pointer member and its initialization to null. The motivation for and the usage of extra data is discussed in Section 5.9.5.

It is critical to understand that the foundation for sharing objects using reference counting is that only one owning RCPNode object can exist for any object that is shared. Consider the code in Listing 66 that creates the reference-counting objects shown in Figure 5. All of these RCP objects share the same RCPNode object.

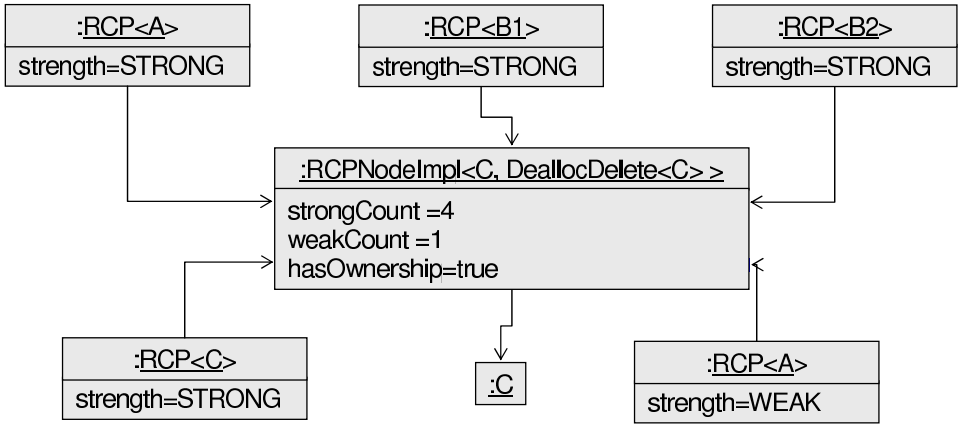



**Figure 5.** Example of several `RCP` objects pointing to the same `RCPNodeImpl` object.

**Listing 66** *: Example of setting up several `RCP` objects pointing to the same reference-counted object shown in Figure 5.*

```
RCP<C> c(new C);
RCP<B1> b1 = c;
RCP<B2> b2 = c;
RCP<A> a1 = c;
RCP<A> a2 = a.create_weak();
```

If the programmer follows the idioms described in Section 5.8 and outlined in Appendix A and Appendix B, it will always be the case that only one reference-counting node object will exist for a given reference-counted object. Exceptions to the one owning `RCPNode` object per reference-counted object guideline are allowed to facilitate some more advanced use cases (see Section 5.13.1 for an example). As mentioned earlier, the `RCPNode` object stores both a strong and a weak reference count. The strong and weak reference counts are equal to the number of strong and weak `RCP` objects pointing to the single `RCP` node object. When the strong count goes to zero, the underlying reference-counted object is destroyed but the `RCPNodeImpl` object is not destroyed until the strong and weak counts both go to zero. The motivation and the workings of strong versus weak reference counts is discussed in Section 5.9.2.

Finally, one of the key integrated debug-mode capabilities of the Teuchos reference-counting machinery is the ability to trace the `RCPNode` objects that are created and destroyed and put them in a low-overhead object database. The `RCPNodeTracer` class/object is a global singleton object that stores all of the `RCPNode` objects in active use. An `std::multimap` object is used to store raw pointers to the `RCPNode` objects and the multi-map is keyed by the `void*` address of the underlying reference-counted objects themselves. Therefore, one can query to see if any `RCPNode` object already exists for a given object. The cost of this query is $O(log(n))$ where $n$ is the number of active `RCPNode` objects currently in use. Therefore, the cost of node tracing quite scalable with the number of `RCPNode` objects in use. The current implementation optionally relies on Boost code which provides some trickery for determining at compile-time if a type is polymorphic or not and thereby allowing the use `dynamic_cast<const void*>(p)` to determine the true base address of any object (no matter if it uses virtual base classes and multiple inheritance or not). The

ability to trace active `RCPNode` objects and look them up based on an object's address is critical for many debug-mode runtime checking capabilities including: a) reporting objects involved in circular dependencies after a program ends (see Section 5.11.2), b) detection of dangling references of non-owning `RCP` objects (see Section 5.11.3), and c) detection of the creation of multiple owning `RCPNode` objects (see Section 5.11.4).

Note that node tracing is only an optional debug-mode feature and is not required for the correct functioning of the reference-counting machinery. In fact, the observable behavior of correct programs is exactly the same whether debug-mode node tracing is enabled or not. For correct programs, the only observable consequence of having node tracing enabled will be increased runtimes.

### 5.9.2 Circular references and weak pointers

The fundamental weakness of low-overhead reference counting as described in this paper and used in the Teuchos (and boost and any other reference-counting) memory management classes is that there is no bullet-proof way to address circular references that otherwise result in memory leaks. Because of possible circular references, only system-level garbage-collection methods, such as implemented in languages like Java and Python, can robustly clean up memory in every case of circular reference. As stated earlier, given backward compatibility constraints, many existing C++ programs cannot be used with any C++ implementation that might implement garbage collection, not now or ever. A key issue is that many programs require the side-effects of the deletion of objects as specific points in the program and changing the time of deletion of the object (and the call of the destructor) would break the program.

To understand the problem with circular references, consider the code in Listing 67 which sets up a simple circular reference between two objects.

**Listing 67** *: Setting up a simple circular reference between two objects*

```
{
  RCP<A> a = createA();
  RCP<B> b = createB();
  a->set_B(b);
  b->set_A(a);
  RCP<ClientA> clientA = createClientA(a);
  RCP<ClientB> clientB = createClientB(b);
  ...
}
// The A and B objects will not be deleted when the above code block ends!
```

The code fragment in Listing 67 sets up the objects in Figure 6 showing the circular reference. Here object `a` contains an `RCP` pointing to object `b`, and object `b` contains an `RCP` pointing to object `a`. In this situation, when `ClientA` and `ClientB` destroy their `RCP` objects pointing to the underlying `a` and `b` objects, the reference counts will not go to zero because of the circular reference between `a` and `b`. This will result in a memory leak that a tool like Valgrind or Purify should complain about. If lots of objects with circular references are constantly being created and destroyed resulting in these types of memory leaks, then obviously one has a problem and the system could run out of memory and bring the program down.

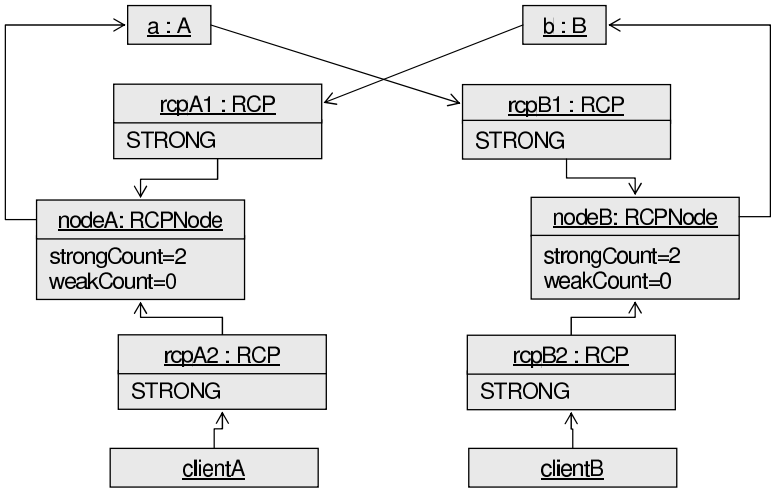



**Figure 6.** Simple circular reference between two objects.

When debug-mode node tracing is enabled and circular references exist, the reference-counting node tracing machinery will print out the remaining `RCPNode` objects when it exists (see Section 5.11.2 for more details).

While there is no completely general and bullet-proof way to address the circular reference problem, there is a fairly simple and cheap approach supported by the Teuchos reference-counting machinery that developers can use to effectively resolve circular references in most cases. The approach described here supported by the Teuchos reference-counted classes is to exploit the concept of weak reference-counted pointers. As shown in Figure 4, this is accomplished through a `strength` attribute with values `STRONG` and `WEAK`. By default, all `RCP` objects are `STRONG`. When an `RCP` is `STRONG`, then the underlying `ConcreteT` object is guaranteed to stay around. However, when the `RCP` is `WEAK`, the underlying `ConcreteT` can get deleted when strong count goes to zero (by the deletion of other `STRONG RCP` objects).

So how can one deal with circular references like this? The answer in this case is to use a weak `RCP` to break the circular reference as shown in Listing 68.

**Listing 68** *: Breaking a simple circular reference using a weak `RCP`*

```
{
  RCP<A> a = createA();
  RCP<B> b = createB();
  a->set_B(b);
  b->set_A(a.create_weak());
  RCP<ClientA> clientA = createClientA(a);
  RCP<ClientB> clientB = createClientB(b);
  ...
  if (deleteClientAFirst)
    clientA = null;
  else
    clientB = null;
```

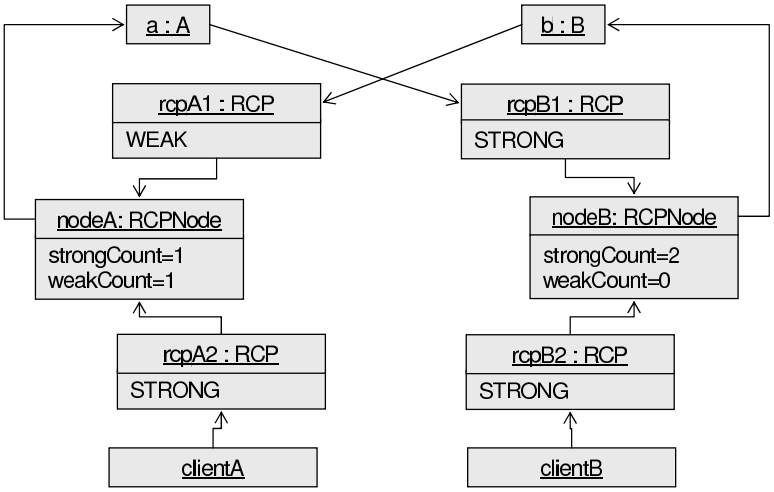



**Figure 7.** Simple circular reference between two objects broken using a `WEAK RCP`.

```
}
// Now all the objects will be deleted correctly no matter if
// clientA or clientB goes away first.
```

The object structure set up by the code in listing Listing 68 is depicted in Figure 7. With the weak pointer in place, all of the objects will get destroyed when `ClientA` and `ClientB` remove their `RCP` objects, no mater what order they remove them. The critical assumption is that the "useful" lifetime of `a` is a super-set of the "useful" lifetime of `b`. If `a` gets deleted before `b`, then `b` had better not try to access `a` anymore! However, the goal is that whoever gets deleted first (i.e. `clientA` or `clientB`), then the objects `a` and `b` will also be deleted gracefully and not result in memory leaks.

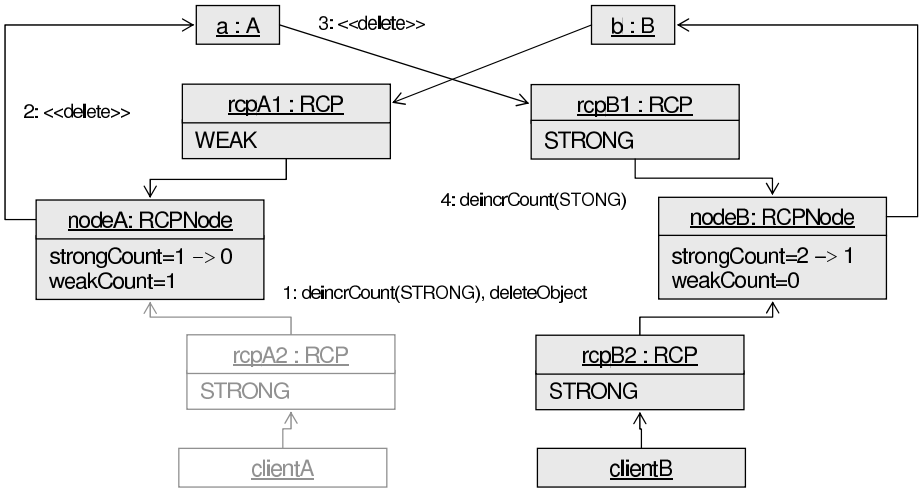
In the next section, detailed scenarios are given for the deletion of the objects shown in Figure 7. This information is important if one wants to understand exactly how the weak pointers can allow the objects to be deleted correctly while still catching mistakes gracefully and avoiding undefined behavior. However, this information is not critical to understand for basic usage of the classes.

### Detailed scenarios for weak pointers and cicular references

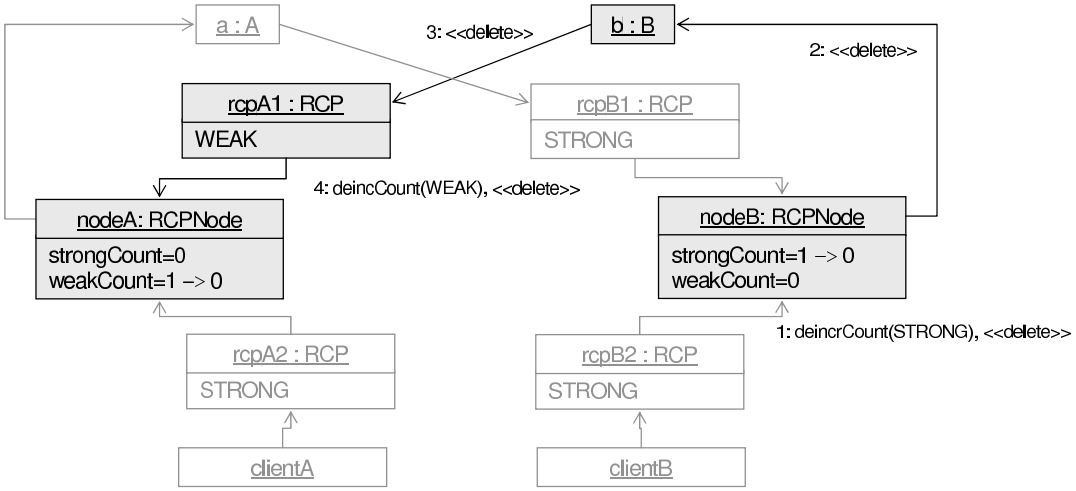
Consider the scenario where the `clientA` object goes away first (i.e. `deleteClientAFirst=true`) depicted in Figure 8 (UML communication diagram). This scenario is shown in two phases in two separate UML communication diagrams in Figure 8 with the following steps:

**a)** `ClientA` goes away first

- **a.1)** As `rcpA2` goes away, it deincrements `nodeA::strongCount` from 1 to 0.
- **a.2)** Since `nodeA::weakCount > 0`, then `nodeA` is not deleted but since `nodeA::strongCount==0` the object `a` gets deleted.




a) `ClientA` goes away first




b) `ClientB` goes away second

**Figure 8.** Weak pointer scenario where `ClientA` is deleted first

**a.3)** As `a` is deleted, it deletes its `RCP` object `rcpB1`.

**a.4)** Since `rcpB1` is a strong pointer, it deincrements `nodeB::strongCount` from 2 to 1. Therefore, neither `nodeB` or `b` gets deleted at this point. NOTE: At this point, the object `a` has been deleted and `nodeA`'s internal pointer has been set to `NULL`. If the `b` object tries to access `a` after this, it will result in an exception being thrown in a debug build. In a non-debug build, any access of `a` from `b` will result in undefined behavior (e.g. segfault).

**b)** `ClientB` goes away second

**b.1)** As `clientB` goes away, it takes `rcpB2` with it. Since `rcpB2` is a strong pointer, it deincrements `nodeB::strongCount` from 1 to 0. Since `nodeB::strongCount` and `nodeB::weakCount` are both 0 this results in `nodeB` being deleted.

**b.2)** As `nodeB` is being deleted, it deletes the `b` object.

**b.3)** As `b` is being deleted, it deletes its `RCP` object `rcpA1`.

**b.4)** With `rcpA1` being deleted it reduces `nodeA::weakCount` from 1 to 0. Since `nodeA::strongCount` and `nodeA::weakCount` are both 0, this results in `nodeA` being deleted. Since the object `a` is already deleted, nothing more happens.

Now consider the scenario where the `clientB` object goes away first (i.e. `deleteClientAFirst=false`) depicted in Figure 9 which involves the following steps:

**a)** `ClientB` goes away first

**a.1)** The `clientB` object goes away first and takes its `RCP` object `rcpB2` with it. This reduces `nodeB::strongCount` from 2 to 1. No other objects are deleted yet.

**b)** `ClientA` goes away second

**b.1)** When `rcpA2` goes away, it deincrements `nodeA::strongCount` from 1 to 0. At this point, since `nodeA::weakCount > 0`, the node is not deleted but the referenced object `a` is deleted.

**b.2)** The object `a` is deleted.

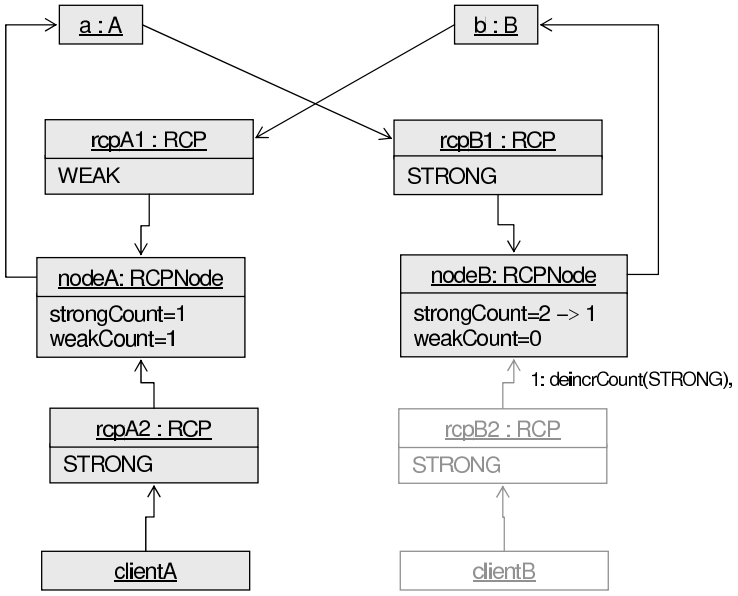
**b.3)** As `a` is deleted, it deletes its `RCP` object `rcpB1`.

**b.4)** As `rcpB1` is deleted, it deincrements `nodeB::strongCount` from 1 to 0. Since `nodeB::weakCount==0`, then `nodeB` is deleted.

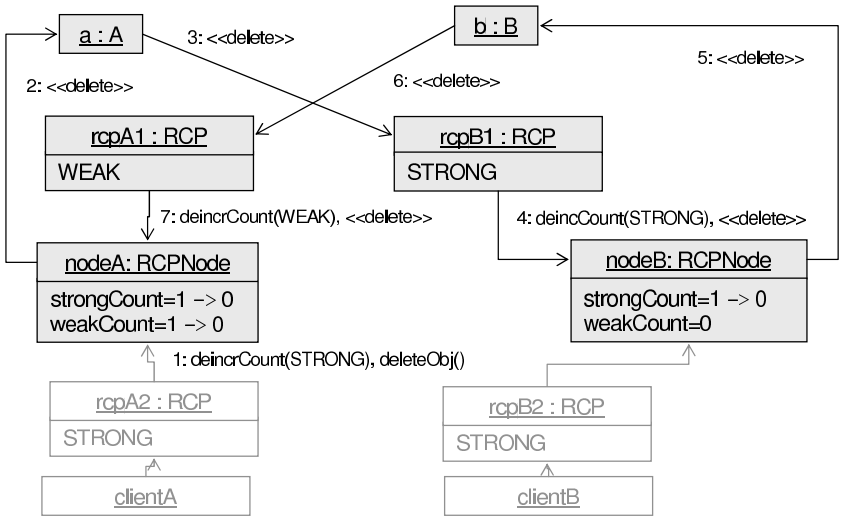
**b.5)** As `nodeB` is deleted, it deletes the `b` object.

**b.6)** As `b` is deleted, it deletes its `RCP` object `rcpA1`. What is critical here is that `b` must not try to access `a` which is already in the process of being deleted. If `b` were to try to access `a` as it is being deleted, in debug mode an exception would be thrown. In non-debug mode, this would result in undefined behavior (e.g. segfault). It is rare, however, that one object tries to access another as they are deleted.

**b.7)** When `rcpA1` is removed, it deincrements `nodeA::weakCount` from 1 to 0. Since `nodeA::strongCount` is already 0, this results in `nodeA` being deleted. Since `a` is already in the process of being deleted, nothing extra happens here.




a) `ClientB` goes away first




b) `ClientA` goes away second

**Figure 9.** Weak pointer scenario where `ClientB` is deleted first

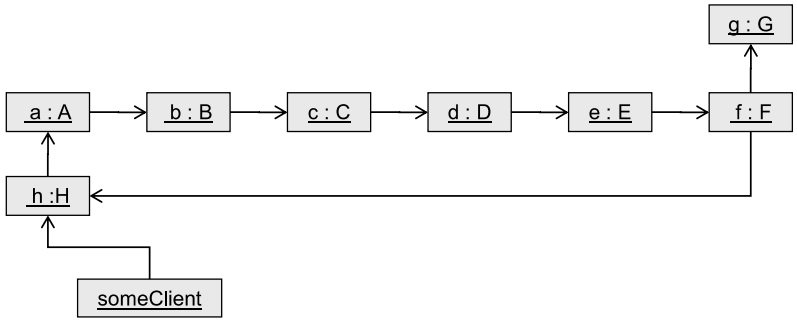



**Figure 10.** Example of a circular chain involving many objects and many classes.

What is especially interesting about the second scenario is how deleting the `a` object in Figure 9.b triggers a chain reaction that causes the `b` object to be deleted which recursively causes the `nodeA` object to be deleted, all in the call stack where the `a` object is being deleted. To accomplish this correctly, the `RCPNodeImpl::deleteObj()` function has some special logic to avoid a double delete call being performed on the reference-counted object.

### Comparison to weak pointers in Boost

With respect the weak pointers, the Teuchos class `RCP` differs substantially from the Boost and therefore the C++0x standard reference-counting classes. With the class `RCP`, the attribution of strong or weak is made at runtime. This allows an external client to decide at runtime to make `a`'s reference to `b` weak or `b`'s reference to `a` weak depending on the given circumstance. With the Boost and C++0x `shared_ptr` class, one has to use a separate class `weak_ptr` to represent a weak pointer. The problem with the Boost approach then is that one has to decide at compile time if a particular reference is going to be weak or strong. While there are some cases where one can always assume the reference needs be weak (like in the self-reference case described in Section 5.13.3), there are more complex cases where one cannot decide this so easily at compile time. For example, if one were to use the `shared_ptr` and `weak_ptr` classes, one would have to decide at compile time to make `a`'s reference or `b`'s reference weak. The decision one makes make might work for one set of use cases that one currently knows about, but for more complex use cases not discovered yet, one may need to switch it. In fact, in the same compiled program there may be some use cases where `a` will be deleted before `b` and other use cases where `b` will be deleted before `a`. With the classes `shared_ptr` and `weak_ptr`, this is impossible to handle (at least not without storing both smart pointer types in each class `A` and `B` object and then using one or the other which is not very elegant or efficient). The only argument for the compile-time approach used by Boost and C++0x is improved performance in both speed and memory overhead but the results in Section 5.12.1 show that this extra overhead is fairly minimal. Overall, the overhead induced by the flexible runtime approach to weak pointers of the `RCP` class (and therefore also the `ArrayRCP` class) is well worth this small extra overhead. Typically, the classes `RCP` and `ArrayRCP` are used to manage objects (or blocks of array data for `ArrayRCP`) much larger than what is contained in the infrastructure for the reference-counting objects so the additional memory overhead is usually insignificant as well.

**Summary of circular references and weak pointers**

While this section has focused on a simple example involving a circular reference between two classes and two objects, in reality circular references typically involve many different objects and classes which may be in very distant parts of the code base which make them very difficult to find by just examining the static code or running the code in a debugger. For example, Figure 10 (UML object diagram) depicts a circular reference involving eight objects. When the external `someClient` object removes its `RCP<H>` object, the chain of objects from `a` to `h` will not be deleted which results in a memory leak. These types of chains of circular references can be very difficult to track down and that is where the debug-mode runtime node tracing described in more detail in Section 5.11.2 comes in most handy.

Section 5.11.3 describes how weak pointers are used in runtime debug checking for dangling (non-persisting) references. Section 5.13.3 describes how weak pointers are used in dealing with object self-references.

In summary, the Teuchos reference-counting machinery dynamically addresses both strong and weak references and is a very powerful tool but to use it effectively, one needs to understand the basic semantics for its use. The good news is that likely 90% of more casual developers who use the classes `RCP` and `ArrayRCP` will never need to know the difference between a strong and weak reference and will by default just use strong references. Weak references will get used under the hood for some debug-mode runtime checking but they are totally transparent to client code and the programmer. It is only in cases of circular references and with some more advanced idioms and design patterns (see Section 5.13.3 and Section 5.13.4 for examples) do typical programmers need to know anything about weak pointers.

### 5.9.3 Customized deallocators

The most common use of `RCP` is to manage the lifetime of objects allocated using operator `new` and deallocated using operator `delete` (or `new []` and `delete []` for `ArrayRCP`). For these use cases, the built-in behavior in `RCP` does exactly the right thing for this average case. However, there are situations when one cannot simply call `delete` to deallocate an object.

Some examples of situations where something other than calling `delete` needs to be performed include when:

1. *Reference counts for objects are managed by requiring clients to explicitly call increment and decrement functions*: This situation occurs when using CORBA [19] and COM [10] for instance. Such an approach is also presented in [25, Item 29] in the subsection "A Reference-Counting Base Class". In these protocols, deallocation occurs automatically behind the scenes when this other reference count goes to zero and does not occur through an explicit call to operator `delete` as with the default behavior for `RCP`.

2. *Objects are managed by certain types of object databases*: In some object databases, an object that is grabbed from the database must be explicitly returned to the database in order to allow proper object deletion to take place later.

3. *A different reference-counted pointer class is used to initially get access to the managed object*: For example, suppose some piece of peer software works with `boost::shared_ptr` (see [7])

referenced-counted objects while the resident software works with `RCP` objects. It then becomes critical no object is deleted until all the clients using either of these smart pointer types remove their references to this underlying object (i.e. by destroying their smart pointer objects or setting them to null).

4. *A C struct object is allocated and deallocated through explicit C function calls*: Here, a C library function must be called to deallocate the object (examples of this exist in unit test and library code).

There are many other additional situations where one cannot simply assume that calling operator `delete` is used to release an object. The bottom line is that in order to be general, one must allow arbitrary policies to be used to deallocate an object after clients are finished using the object.

Perhaps the key differentiating property between a flexible high quality reference-counted pointer implementation and a poor implementation is the capability to allow the user to define an arbitrary deallocator policy that defines exactly what it means to release a reference-counted object (or array of objects). The reference-counted Teuchos classes `RCP` and `ArrayRCP`, as well as `boost::shared_ptr` all allow the client to specify a user-defined deallocation policy object when the first reference-counted object is constructed.

The code associated with customized deallocation policies for `RCP` (which are also identical for `ArrayRCP`) are shown in Listing 69.

**Listing 69** *: Declarations for customized deallocation policies for `RCP`*

```
// Default decallocation policy for RCP
template<class T>
class DeallocDelete
{
public:
  typedef T ptr_t;
  void free( T* ptr ) { if(ptr) delete ptr; }
};

// Other provided deallocation policy classes
template<class T> class DeallocNull { ... };
template<class T> class DeallocArrayDelete { ... };
template<class T> class DeallocFunctorDelete { ... };
template<class T> class DeallocFunctorHandleDelete { ... };
template<class T> class EmbeddedObjDealloc { ... };

template<class T>
class RCP {
public:
  ...
  template<class Dealloc_T>
  RCP(T* p, Dealloc_T dealloc, bool has_ownership);
  ...
};

// Non-member constructors for deallocators and extraction functions
```

```
template<class T, class Dealloc_T>
RCP<T> rcp(T* p, Dealloc_T dealloc, bool owns_mem);

template<class Dealloc_T, class T>
const Dealloc_T& get_dealloc(const RCP<T>& p);

template<class Dealloc_T, class T>
Dealloc_T& get_nonconst_dealloc(const RCP<T>& p);

template<class Dealloc_T, class T>
Ptr<const Dealloc_T> get_optional_dealloc(const RCP<T>& p);

template<class Dealloc_T, class T>
Ptr<Dealloc_T> get_optional_nonconst_dealloc(const RCP<T>& p);
```

All deallocator objects must support the typedef member `ptr_t` and function member `free(...)`. The concept of a template policy interface (also called a function object [29, Section 18.4]) should hopefully be familiar to semi-advanced users of the STL (part of the standard C++ library).

To demonstrate the use of a deallocator object, let us assume that the code must wrap objects of type `A` managed by the object database shown in Listing 70.

**Listing 70** *: Example of a simple object database*

```
class ObjectADB {
  ...
  A& get(int id);
  void release(int id);
  ...
};
```

In the object database in Listing 70, objects are accessed and released using an integer ID. How this ID is specified and determined is not important here. Let us suppose that one wants to define an abstract factory that returns objects of type `A` wrapped in `RCP<A>` objects using a database object of type `ObjectADB` shown in Listing 70. For this abstract factory, objects of type `A` will be allocated from a list of ids given to the factory. The outline of this abstract factory subclass is shown in Listing 71:

**Listing 71** *: Factory subclass that allocates new objects using an `ObjectADB` object*

```
class ObjectADBFactory : public AbstractFactory<A> {
  RCP<ObjectjADB> db_;
  Array<int> ids_;
public:
  ObjectADBFactory(const RCP<ObjectADB>& db, const ArrayView<const int>& ids)
   : db_(db), ids_(ids) {}
  RCP<A> create();  // Overridden from AbstractFactory
};
```

The above abstract factory subclass `ObjectADBFactory` inherits from a generic `AbstractFactory` base class that defines a pure virtual method `create()`. In order to implement the `create()` function, a deallocator class must be defined and used shown in Listing 72.

**Listing 72** *: Custom deallocator class for releasing objects managed by `ObjectADB`*

```
class DeallocObjectADB
{
  RCP<ObjectjADB> db_;
  int id_;
public:
  DeallocObjectADB(const RCP<ObjectjADB>& db, int id)
    : db_(db), id_(id) {}
  typedef A ptr_t;
  void free(A* ptr) { db_->release(id_); }
};
```

Now one can define the implementation of the `create()` function override as shown in Listing 73.

**Listing 73** *: Implementation of the factory create function*

```
RCP<A> ObjectADBFactory::create()
{
  TEST_FOR_EXCEPTION(ids_.size()==0, std::runtime_error, "No ids are left!");
  const int id = ids_.pop();
  return rcp(&db_->get(id), DeallocObjectADB(db_, id), true);
}
```

The program in Listing 74 shows the use of the factory subclass `ObjectADBFactory` defined in Listings 70, 71, 72, and 73.

**Listing 74** *: Example driver program that transparently uses the `ObjectADBFactory` class*

```
int main()
{
  // Create the object database and populate it (and save the ids)
  RCP<ObjectADB> db;
  Array<int> ids;
  ...
  // Create the abstract factory object
  ObjectADBFactory  ftcy(db, ids());
  // Create some A objects and use them
  RCP<A> a_ptr1 = fcty.create();
  ...
  return 0;
}
```

In the example program in Listing 74, all of the objects of type `A` are created and removed seamlessly without the client code that interacts with `RCP` and `AbstractFactory` knowing anything about what is going on under the hood.

Examples of other types of deallocators are given in the unit test suite for the `RCP` class.

### 5.9.4 Embedded objects

Support for customized template deallocator policy objects described in Section 5.9.3 turns out to be a pretty flexible feature. The ability to embed any arbitrary object in the `RCPNode` object gives one an efficient way to define a different deallocation policy that is invoked by the destructor on the object instead of requiring an explicit deallocation policy object. In addition, one can also tack on any extra data desired and embed it in the underlying `RCPNodeImpl` object. The only restriction is that one has to make the choice of what to embed in the `RCPNode` object when the very first `RCP` object is created (which in turn creates the concrete templated `RCPNodeImpl` object). If one wants the flexibility to embed other data in the underlying `RCPNode` object after it has been created then the "extra data" feature needs to used which is described in Section 5.9.5. The advantage of embedding objects in the deallocator in the `RCPNodeImpl` object is that it can be quite a bit more efficient than using the "extra data" feature which requires more runtime-support and greater overhead.

The functions that are used to embed objects when creating `RCP` objects and retrieve them again are shown in Listing 75 (identical functions exist for the `ArrayRCP` class).

**Listing 75** *: Embedded object functions for RCP*

```
template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObjPreDestroy(T* p, const Embedded &embedded,
  bool owns_mem=true);

template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObjPostDestroy(T* p, const Embedded &embedded,
  bool owns_mem=true);

template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObj(T* p, const Embedded &embedded, bool owns_mem=true);

template<class TOrig, class Embedded, class T>
const Embedded& getEmbeddedObj(const RCP<T>& p);

template<class TOrig, class Embedded, class T>
Embedded& getNonconstEmbeddedObj(const RCP<T>& p);

template<class TOrig, class Embedded, class T>
Ptr<const Embedded> getOptionalEmbeddedObj( const RCP<T>& p );

template<class TOrig, class Embedded, class T>
Ptr<Embedded> getOptionalNonconstEmbeddedObj( const RCP<T>& p );
```

The embedded object functions in Listing 75 simply use the custom templated deallocator class `EmbeddedObjDealloc` shown in Listing 76 along with the public deallocator functions in Listing 69.

**Listing 76** *: RCP Deallocator using an embedded object*

```
template<class T, class Embedded, class Dealloc>
class EmbeddedObjDealloc
{
public:
  typedef typename Dealloc::ptr_t ptr_t;
  EmbeddedObjDealloc(
    const Embedded &embedded, EPrePostDestruction prePostDestroy,
    Dealloc dealloc
    ) : embedded_(embedded), prePostDestroy_(prePostDestroy), dealloc_(dealloc)
    {}
  void setObj( const Embedded &embedded ) { embedded_ = embedded; }
  const Embedded& getObj() const { return embedded_; }
  Embedded& getNonconstObj() { return embedded_; }
  void free( T* ptr )
    {
      if (prePostDestroy_ == PRE_DESTROY)
        embedded_ = Embedded();
      dealloc_.free(ptr);
      if (prePostDestroy_ == POST_DESTROY)
        embedded_ = Embedded();
    }
private:
  Embedded embedded_;
  EPrePostDestruction prePostDestroy_;
  Dealloc dealloc_;
  EmbeddedObjDealloc(); // Not defined and not to be called!
};
```

The customized deallocator class in Listing 76 is then templated with `DeallocDelete` (see Listing 69) and set by the non-member constructor functions in Listing 75. The distinction between pre- and post-destroy can be critical depending on how the embedded data is used (many examples are given in this paper). In most cases, the order the embedded object is reset to the default value is not important and therefore the client would just use `rcpWithEmbeddedObj(...)` to set the embedded object (in which case it uses post-destruction by default).

Typically, the embedded object will be some `RCP` such that when the embedded object is assigned to the default state as in `embedded_ = Embedded()` then the destructor on that object will be called (which is what happens when the strong count goes to zero with `RCP`). A simple example of embedding an `RCP` that controls memory release is shown in Listing 77:

**Listing 77** *: A simple example of using embedded objects*

```
RCP<A> a_ptr1(new A);
RCP<A> a_ptr2 = rcpWithEmbeddedObj(a_ptr1.getRawPtr(), a_rcp1, false);
```

What the code in Listing 77 does it is creates a new now-owning `RCPNodeImpl` object with an `RCP` object embedded in it. This maintains the correct ownership semantics by resets the reference count in the new `RCPNodeImpl` object. The use case shown in Listing 77 may look silly and trivial but it is the foundation for several more advanced use cases (see Section 5.13.1 for a related example). As a result of this code, the underlying `A` object will not be deleted until the `RCPNodeImpl` object associated with `a_ptr2`, and all of the `RCP` objects created from it, are destroyed. Even the above simple use case can be useful if one wants to be able to use the reference count on `RCP` objects derived from `a_ptr2` to determine usage of the object by other clients. There are concrete examples of this exact simple usage in production code.

A more general usage of embedded objects to perform arbitrary actions is demonstrated in the context of the "generalized view" design pattern in Section 5.13.4.

### 5.9.5 Extra data

As mentioned in Section 5.9.1, the Teuchos reference-counting machinery supports storing and retrieving arbitrary objects as extra data stored on the `RCPNode` object itself. The functions supporting extra data for the `RCP` class are shown in Listing 78 (the functions for `ArrayRCP` are identical).

**Listing 78** *: `RCP` extra data functions*

```
template<class T1, class T2>
void set_extra_data(const T1 &extra_data, const std::string& name,
  const Ptr<RCP<T2> > &p, EPrePostDestruction destroy_when = POST_DESTROY,
  bool force_unique = true);

template<class T1, class T2>
const T1& get_extra_data(const RCP<T2>& p, const std::string& name);

template<class T1, class T2>
T1& get_nonconst_extra_data(RCP<T2>& p, const std::string& name);

template<class T1, class T2>
Ptr<const T1> get_optional_extra_data(const RCP<T2>& p, const std::string& name);

template<class T1, class T2>
Ptr<T1> get_optional_nonconst_extra_data(RCP<T2>& p, const std::string& name);
```

Given the support for embedded objects described in Section 5.9.4, extra data rarely needs to be used. Embedding and retrieving objects in the templated `RCPNodeImpl` object is more efficient that using the more general `std::map` object and `any` wrapper that are used to implement the "extra data" feature and therefore embedded objects should be used whenever possible instead of extra data. However, there are a few key advantages to using extra data over embedded objects that may be worth the performance overhead or using extra data may be the only way to address an issue and some examples include:

- *One can associate new extra data after the RCPNode object is created.* With embedded objects, one can only select the data-type for the embedded object at the time when the first `RCP` object is created.

- *One can retrieve data without having to know the concrete template types in the `RCPNodeImpl` object.* With extra data, one only needs to know the string name and the type of the extra data that needs to be retrieved. With embedded objects, the original type of the underling reference-counted object that is used to template the `RCPNodeImpl` class also needs to be known (to see this compare the template arguments for the `getEmbeddedObj(...)` and `get_extra_data(...)`). If this type changes (i.e. if the creating code changes the subclass implementation `TOrig` used), then this will break client code that tries to retrieve the embedded object. Therefore, client code that retrieves embedded object data is more fragile than code that retrieves extra data.
- *One can completely change the deallocation policy at runtime after the RCPNode object has been created.* With embedded objects, the deallocation policy of a reference-counted object cannot be changed after the initial `RCPNodeImpl` object has been created; with extra data it can.

To demonstrate the power and flexibility of extra data, let's consider a (perhaps unlikely) scenario where some piece of code incorrectly associates the wrong deallocation policy to an allocated object shown in Listing 79.

**Listing 79** *: Example of incorrect deallocator*

```
RCP<A> createRCPWithBadDealloc()
{
  return rcp(new A[1]); // Will use delete but should use delete []!
}
```

Hopefully no one would write code like is shown in Listing 79 (but shockingly I did once write code similar to this). However, let's suppose that one has to use the function `createRCPWithBadDealloc()` to allocate `A` objects and are stuck with a pre-compiled library and one cannot access the source code to fix the problem. On most systems an error like this will be tolerated and not cause problems but tools like Valgrind and Purify will complain about code like this to no end and there may be some platforms where this will actually cause the program to crash (since this has undefined behavior).

With `RCP` and extra data, one can replace the deallocation policy on the fly to use the correct policy. The first step is to create a class that will call `delete []` on the pointer correctly as shown in Listing 80.

**Listing 80** *: Deallocator class for extra data deallocation*

```
template<typename T>
class DeallocArrayDeleteExtraData {
public:
  static RCP<DeallocArrayDeleteExtraData<T> > create(T* ptr)
    { return rcp(new DeallocArrayDeleteExtraData(ptr)); }
  ~DeallocArrayDeleteExtraData() { delete [] ptr_; }
private:
  Ptr<T> ptr_;
  DeallocArrayDeleteExtraData(T* ptr) : ptr_(ptr) {}
};
```

The client code can then fix the deallocation policy as shown in Listing 81.

**Listing 81** *: Using* `DeallocArrayDeleteExtraData` *as extra data to fix deallocation policy*

```
// Create object with bad deallocator
RCP<A> a = createRCPWithBadDealloc();

// Disable default (incorrect) dealloc and set a new deallocation policy as extra data!
a.release();
set_extra_data( DeallocArrayDeleteExtraData<A>::create(a.getRawPtr()),
  "dealloc", inOutArg(a));
```

The kind of flexibility shown in the above example is not possible using embedded objects and is not possible with classes like `boost:shared_ptr`. There are numerous other uses for extra data to fix nasty memory management problems (which is why the extra data feature was added in the first place). However, in well designed software, there is no need for a feature like this so a developer should count themselves lucky if they never need to use the extra data feature.

## 5.10 Roles and responsibilities for persisting associations: factories and clients

There are two fundamentally different sets of actors that play two different roles in the use of the reference-counted classes used for persisting associations: a) factory entities that first create the reference-counted object `RCP<A>` and defines the deallocation policy, and b) general clients that accept and use a shared reference-counted object `A` through an `RCP<A>` object.

Factory entities first create the reference-counted object (or array) and construct the first `RCP` (or `ArrayRCP`) object containing it. The most basic type of factories are non-member constructor functions described in Section 5.8.1. When the first `RCP` object is created, the factory gets to decide exactly how object (or array) will be released when the strong reference count goes to zero. The default behavior, of course, is to just simply call `delete` (or `delete []` for arrays) on the contained raw pointer. However, the factory can also choose any arbitrary action imaginable to occur when the reference-count goes to zero. This is set up using a template deallocator policy object as described in Section 5.9.3.

Alternatively, the responsibilities of general clients that use and share a reference-counted object are very simple and these responsibilities are:

- Accept the persisting relationship for a shared reference-counted object through an `RCP` object (or `ArrayRCP` for arrays) as described in Section 5.8.4.
- Share the reference-counted object with other clients by creating a copy of one's `RCP` (or `ArrayRCP`) object and giving it to them.
- When one is finished using the object, simply delete or set to null all of one's `RCP` objects. If some other client is still using the object, it will remain alive. If the client's is the last (strong) reference, then the deallocator policy object that is embedded in the underlying `RCPNodeImpl` object is invoked which knows exactly how to clean up and reclaim the underlying object (or array of memory).

That is all there is to it. Factories create the underlying object(s) wrapped in the first `RCP` object and define how the referenced object(s) will be reclaimed when it is time to do so. General clients just accept and

maintain their references to shared objects (or arrays) by accepting and storing `RCP` objects (or `ArrayRCP` objects) and then setting them to null when they are finished using the object(s).

## 5.11 Debug-mode runtime checking

The primary reason that these Teuchos memory management classes need to be developed in tandem with each other and know each other's internal implementations to some extend is to be able to implement robust and effective debug-mode runtime checking. The debug-mode runtime testing that is built into these classes is very strong and will catch nearly every type of programmer error that is possible, as long as raw C++ pointers are never externally exposed and if raw C++ references are only used for persisting associations. The different categories of debug-mode runtime testing are described in the following subsections along with what the typical diagnostic error messages look like that are attached to exceptions when they are thrown.

### 5.11.1 Detection of null dereferences and range checking

One of the most basic types of debug-mode runtime checking performed by the Teuchos memory management classes are for attempts to dereference a null pointer and range checking of arrays and iterators.

**Listing 82** *: Debug-mode null dereference checking (all types)*

```
RCP<A> a_ptr;        // Default constructs to null
A &a_ref = *a_ptr;   // Throws!
a_ptr->someFunc();   // Throws!

ArrayRCP<int> aa;        // Default constructs to null
a[0];                    // Throws!
int &i_ref = *a.begin(); // Throws!

...
```

All of the Teuchos memory management classes throw on null dereferences. While most systems will abort the program on null dereferences there are some platforms (e.g. some Intel C++ compilers) that will not and it will result in memory errors that may not be seen until later in the program. Technically, dereferencing a null pointer has undefined behavior and compilers and runtime systems can do anything they want with undefined behavior (including corrupting memory and continuing as is the case with some Intel C++ compilers).

The Teuchos array classes `Array`, `ArrayView`, `ArrayRCP`, and `Tuple` all perform array bounds checking in debug-mode builds:

**Listing 83** *: Debug-mode array-bounds checking (all Teuchos array types)*

```
Array<int> a(n);
a[-1];   // Throws!
a[n];    // Throws!
```

In a debug-mode build of the code, all the iterators returned by the `begin()` and `end()` functions of the classes `Array`, `ArrayView`, `ArrayRCP`, and `Tuple` are of the type `ArrayRCP` which is a fully ranged checked iterator.

**Listing 84** *: Debug-mode iterator range checking (all Teuchos array types)*

```
Array<int> a(n);
*(a.begin()-1);           // Throws!
*(a.begin() + a.size());  // Throws!
*a.end();                 // Throws!
```

In addition, comparisons between iterators will thrown if they do not point into the same underlying contiguous array of memory.

**Listing 85** *: Debug-mode iterator matching checking (all Teuchos array types)*

```
ArrayRCP<int> a_arcp = arcp<int>(n);
Array<int> a(n);
// Simple mistake calling standard STL algorithm
std::copy( a.begin(), a_arcp.begin(), a_arcp.end() ); // Throws!
```

These types of checks are fairly straightforward but are extremely useful and work on every platform. This checking is built into programs automatically in a debug-mode build of the code. Contrast this to checked STL implementations that may or may not exist on a given platform and if they do exist, the quality of the implementations can vary widely. Note that in a non-debug build of the code, none of these checks are performed which leads to the fastest code possible.

### 5.11.2 Detection of circular references

One of the more sophisticated types of debug-mode runtime checking supported by the Teuchos memory management classes is the detection and reporting of circular `RCP` references that result in memory leaks. The issue of circular references and the concept of weak pointers was outlined in Section 5.9.2. When debug-mode node tracing is enabled, the reference-counting machinery keeps track of all the `RCPNode` objects that are created and destroyed. If the program ends and there are one or more `RCPNode` objects that are still remaining, then a error message is printed to `std::cerr` that gives all the details of the objects involved in the circular reference.

For example, consider the simple circular reference created in Listing 67 and shown in Figure 6. If left this way, when debug-mode node tracing is enabled, the program ends and prints an error message like the following to `std::cerr`:

**Listing 86** *: Example error message printed after a program ends when there are unresolved strong circular references*

```
***
*** Warning! The following Teuchos::RCPNode objects were created but have
*** not been destroyed yet.  A memory checking tool may complain that these
*** objects are not destroyed correctly.
***
*** There can be many possible reasons that this might occur including:
***
***   a) The program called abort() or exit() before main() was finished.
***      All of the objects that would have been freed through destructors
***      are not freed but some compilers (e.g. GCC) will still call the
***      destructors on static objects (which is what causes this message
***      to be printed).
***
***   b) The program is using raw new/delete to manage some objects and
***      delete was not called correctly and the objects not deleted hold
***      other objects through reference-counted pointers.
***
***   c) This may be an indication that these objects may be involved in
***      a circular dependency of reference-counted managed objects.
***

  0: RCPNode (map_key_void_ptr=0x4a3ff50)
       Information = {T=A, ConcreteT=A, p=0x4a3ff50, has_ownership=1}
       RCPNode address = 0x4a3ffa8
       insertionNumber = 23
  1: RCPNode (map_key_void_ptr=0x4a40548)
       Information = {T=B, ConcreteT=B, p=0x4a40548, has_ownership=1}
       RCPNode address = 0x4a405f0
       insertionNumber = 24

NOTE: To debug issues, open a debugger, and set a break point in the function where the
the RCPNode object is first created to determine the context where the object first
gets created.  Each RCPNode object is given a unique insertionNumber to allow setting
breakpoints in the code.  For example, in GDB one can perform:

1) Open the debugger (GDB) and run the program again to get updated object addresses

2) Set a breakpoint in the RCPNode insertion routine when the desired RCPNode is first
inserted.  In GDB, to break when the RCPNode with insertionNumber==3 is added, do:

  (gdb) b 'Teuchos::RCPNodeTracer::addNewRCPNode( [TAB] [ENTER]
  (gdb) cond 1 insertionNumber==3 [ENTER]

3) Run the program in the debugger.  In GDB, do:

  (gdb) run [ENTER]

4) Examine the call stack when the prgoram breaks in the function addNewRCPNode(...)
```

This error message is enough information to allow one to open a debugger, and set a break-point in the function `RCPNodeTracer::addNewRCPNode(...)` and then examine where these objects are getting created that result in the circular reference (see Section 5.11.7).

Note that in reality, the circular references will involve many objects (sometimes more than a dozen as shown in Figure 10) and therefore this output will contain many `RCPNode` objects. A program may also contain large numbers of smaller sets of circular dependencies. One example in Trilinos had a test that generated hundreds of thousands of smaller circular cycles and leaked memory from hundreds of thousands of objects.

### 5.11.3 Detection of dangling references

Another useful and necessary form of debug-mode runtime checking involves the detection and reporting of access to invalid objects and arrays made through dangling references. A dangling reference is a catch-all term that refers to any pointer or reference that points to a no-longer valid object or array. For example, the following code fragment shows invalid access to a dangling iterator to an array that has changed shape:

**Listing 87** *: Example of a dangling iterator*

```
Array<int> a(n);
Array<int>::iterator itr = a.begin();
a.resize(0);
*itr = 1; // Invalid access of dangling iterator (throws)!
```

In debug-mode, the above example would result in an exception being thrown with an error message like shown below:

**Listing 88** *: Example of a dangling reference error message*

```
Teuchos_RCPNode.hpp:515:

Throw number = 3

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 underling object has already
been deleted since the strong count has already gone to zero.

Context information:

  RCP type:             Teuchos::ArrayRCP<int>
  RCP address:          0x7fbfffec98
  RCPNode type:         Teuchos::RCPNodeTmpl<int,
    Teuchos::EmbeddedObjDealloc<int,
      Teuchos::RCP<__gnu_debug_def::vector<int, std::allocator<int> > >,
      Teuchos::DeallocArrayDelete<int> > >
  RCPNode address:      0xab65a0
  insertionNumber:      5
  RCP ptr address:      0xab4c50
  Concrete ptr address: 0xab4c50

NOTE: To debug issues, open a debugger, and set a break point in the function where the
```

```
the RCPNode object is first created to determine the context where the object first
gets created. ...
```

The erorr message shown in Listing 88 contains all the information needed to open a debugger, run the program again to create new pointer addresses, set up breakpoints and break conditions, and debug the problem. Breakpoints can be set when the `RCPNode` object is first created and inserted and also when the exception is thown (see Section 5.11.7). The NOTE at the bottom of the error message in Listing 88 is really the same as shown in Listing 86 and is only cut off to save space.

A few other examples of dangling references are shown in Listings 89–90.

**Listing 89** *: Example of a dangling `ArrayView`*

```
ArrayView<int> av;
{
  Array<int> a(n);
  av = a;
}
av[0] = 1; // Invalid access to dangling ArrayView (throws)
```

**Listing 90** *: Example of a dangling `Ptr`*

```
Ptr<A> a_ptr;
{
  RCP<A> a_rcp = createA();
  a_ptr = a_rcp.ptr();
}
a_ptr->someFunction(); // Invalid access to dangling Ptr (throws)
```

In general, `Ptr`, `ArrayView` and iterators (returned from the `begin()` member functions) all can be involved in dangling references. Therefore, anytime a `Ptr`, `ArrayView`, or iterator object is created from some other Teuchos memory management object, one can expect that in a debug build that dangling references will checked for and if detected will result in exceptions being thrown with very detailed error messages like shown in Listing 88.

Note that the ability to detect a dangling `ArrayView` of an `Array` object as shown in Listing 89 is due to the fact that the debug-mode internal implementation of `Array` is designed to support this. Compare this to `ArrayView` views of `std::vector`, as shown in Listing 91 where dangling references cannot be detected.

**Listing 91** *: Dangling `ArrayView` of `std::vector` (cannot detect dangling references)*

```
ArrayView<int> av;
{
  std::vector<int> v(n);
  av = v;
}
av[0] = 1; // Invalid access to dangling ArrayView (does *not* throw)
```

The code in Listing 91 has undefined behavior and will most likely segfault if one is lucky. If unlucky, the program may actually appear to run correctly on the main development and testing platforms and it will not be until moved to a production platform that the ill-effects of this erroneous code will be seen. This is one example of why it is so important to use `Array` instead of raw `std::vector` objects. Strong debug-mode runtime checking of `ArrayView` views are not possible when using `std::vector`.

Another type of more sophisticated debug-mode dangling reference detection involves non-owning `RCP` objects to existing reference-counted objects. Consider code like shown in Listing 92.

**Listing 92** *: Example of a dangling non-owning `RCP` object detected through node tracing*

```
RCP<A> a_rcp = createA();
A &a_ref = *a_rcp;
RCP<A> a_rcp2 = rcpFromRef(a_ref);  // Same as rcp(a_ref.getRawPtr(), false)
a_rcp = null;   // The 'A' object gets deleted (a_rcp2 is a dangling pointer)
a_rcp2->someFunction();  // Invalid reference to deleted 'A' object (throws)
```

In a debug-mode build with node tracing turned on, the dangling non-owning `RCP` reference `a_rcp2` in Listing 92 will be caught by the system. This works because the statement `rcpFromRef(a_ref)` results in a call to `RCPNodeTracer::getExistingRCPNode(...)` to look-up the existing `RCPNode` object that points to the same `A` object. In this case, the existing `RCPNode` object is found and it is used to create a weak `RCP` object (see Section 5.9.2) that can then detect if the original reference-counted object has been deleted. Again, this more sophisticated type of debug-mode runtime checking requires that node tracing be enabled[12].

Debug-mode runtime detection and reporting of dangling references is built on the foundation of weak `RCP` and `ArrayRCP` objects. Basically, all non-persisting views use a weak `RCP` or `ArrayRCP` object (see Section 5.9.2) internally to allow the parent object to be changed or be deleted and to detect this if a client tries to access the now invalid object through the dangling reference.

### 5.11.4 Detection of multiple owning `RCP` objects

There are other types of invalid usage that can be detected and caught in debug-mode with node tracing enabled. Consider, for example, what happens when one or more of the commandments in Appendix B are broken and more than one owning `RCPNode` object is created pointing to the same underlying `ConcreteT` object as shown in Figure 11 generated by the sloppy code shown in Listing 93.

**Listing 93** *: Invalid creation of dual owning `RCPNodeImpl` objects (shown in Figure 11)*

```
C *c_raw_ptr = new C;
RCP<C> c_ptr(c_raw_ptr);
```

[12] In order to handle multiple inheritance and virtual bases classes and still get the correct base object address, Boost support must also be configured which is needed to use `boost::is_polymorphic` to allow the use of `dynamic_cast<void*>(...)` to determine the true base address of a polymorphic object. Otherwise, without this, the system will not be able to determine if two abstract interfaces really point to the same object and therefore the look-up of the `RCPNode` object may fail to detect when two addresses are pointing to the same object.

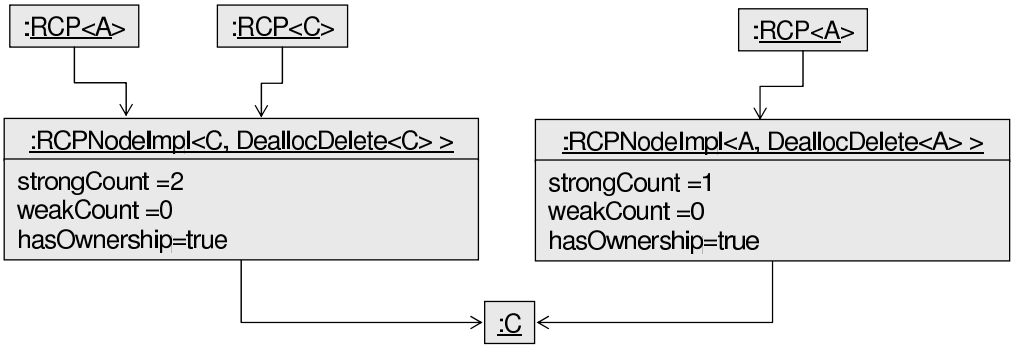



**Figure 11.** Example of duplicate owning `RCPNodeImpl` objects

```
RCP<A> a_ptr1 = c_ptr;
...
A *a_raw_ptr = c_raw_ptr;
RCP<A> a_ptr2(a_raw_ptr);
```

The problem is that the two `RCPNodeImpl` objects generated by Listing 93 (shown in Figure 11) do not know about each other and the first one who has its strong reference count go to zero will result in the underlying `C` object being deleted. In this case, the other remaining `RCPNodeImpl` object, and all of the resulting `RCP` objects pointing to it will be left with a non-null pointer to a now deleted `C` object. If the client tries to access the underlying object through one of these now invalid references, it will yield undefined behavior and will likely result in a segfault (if one is lucky). Also, a second call to `delete` will also occur even if invalid access is not performed.

Not to fear, in a debug-mode build with node tracing enabled, the `RCPNodeTracing` object automatically detects the creation of the second owning `RCPNodeImpl<A>` object and will thrown an exception with an error message that looks something like Listing 94.

**Listing 94** *: Example of an error message from a the attempt to create dual owning `RCPNodeImpl` objects*

```
Trilinos/packages/teuchos/src/Teuchos_RCPNode.cpp:240:

Throw number = 1

Throw test that evaluated to true: rcp_node_already_exists && rcp_node->has_ownership()

RCPNodeTracer::addNewRCPNode(rcp_node): Error, the client is trying to create a new
RCPNode object to an existing managed object in another RCPNode:

  New RCPNode {address=0x9cb3e0, base_obj_map_key_void_ptr=0x9cac40,
    base_obj_type_name=A, map_key_void_ptr=0x9cac40, has_ownership=1, insertionNumber=6}

  Existing RCPNode {address=0x9cb2b0, base_obj_map_key_void_ptr=0x9cac40,
   base_obj_type_name=C, map_key_void_ptr=0x9cac40, has_ownership=1, insertionNumber=5}
```

```
  Number current nodes = 6

This may indicate that the user might be trying to create a weak RCP to an existing
object but forgot make it non-ownning.  Perhaps they meant to use rcpFromRef(...)
or an equivalent function?

NOTE: To debug issues, open a debugger, and set a break point in the function where the
the RCPNode object is first created to determine the context where the object first
gets created. ...
```

A debugger can be opened, a break-point can be set in the function `TestForException_break(...)`, and the program can be run again and break at the time the exception is thrown to see the context under which the second illegal `RCPNode` is created (see Section 5.11.7). A breakpoint can also be set in the function `addNewRCPNode(...)` to see when the other `RCPNode` object was created (see Section 5.11.7).

If one is willing to pay for a little extra overhead of `RCPNode` tracing (see Section 5.12.1 for some timing results of the overhead), then node tracing will detect the erroneous creation of multiple owning `RCPNode` objects and respond in a graceful way. Note that creating multiple non-owning `RCPNode` objects is okay and is allowed both when node tracing is enabled and when it is not enabled (however, see Commandment 5 in Appendix B for restrictions on the creation of owning and non-owning `RCP` objects).

### 5.11.5 Performance of debug-mode checking versus memory checking tools

One of the common criticisms of debug-mode runtime checking is that it incurs an unacceptably large runtime overhead. However, this overhead is only incurred for debug-mode builds and does not affect non-debug optimized builds. To speed up debug-mode runtime checking, one can compile with optimized compiler options (e.g. `-O3`) which significantly speeds up the code. Also, one has to consider the relative cost of built-in debug-mode runtime checking versus running a memory checking tool like Valgrind or Purify.

To investigate the cost of debug-mode runtime checking, the Trilinos package Tpetra[13] is used since it relies the Teuchos memory management classes at a very low level and therefore would be expected to show the largest runtime overhead for debug-mode checking. Table 16 shows the runtime of the Tpetra serial test suite (12 test programs) for several different build and runtime configurations. In all of these builds, optimized compiler options (-O3) were used. All of these timing tests were performed on an a 3.2GHz AMD machine with 8 cores running Linux 2.6.9-78.0.1.ELsmp using GCC 3.4.6. Valgrind tests were run using version 3.2.1. All of the test executables were run in serial on the unloaded Linux machine.

The results in Table 16 give the total runtimes as well as the relative runtimes for debug-mode checking and Valgrind. The second column 'Runtime' gives the raw CPU time in seconds (as reported by CTest) for all 12 test executables in the Tpetra test suite. The third column 'Multiplier' gives the ratio of the runtime relative to the base-line optimized build case. The fourth column 'Valgrind Multi' gives the fractional increase in the runtime of the test suite run with Valgrind relative to running the same executables without Valgrind.

The results in Table 16 show that while the cost incurred by debug-mode runtime checking can be

[13] `http://trilinos.sandia.gov/packages/tpetra/`

| Configuration | Runtime (sec) | Multiplier | Valgrind Mult |
|---|---|---|---|
| 1) Optimized build (base-line) | 0.16 | 1.0 | - |
| 2) Debug-mode runtime checking | 0.49 | 3.1 | - |
| 3) Debug-mode runtime checking + node tracing | 1.08 | 6.8 | - |
| 4) Valgrind optimized build | 56.21 | 351.3 | 351.3 |
| 5) Valgrind debug-mode runtime checking | 214.01 | 1337.6 | 431.5 |
| 6) Valgrind debug-mode runtime checking + node tracing | 378.54 | 2365.9 | 347.9 |

**Table 16.** Overhead of runtime checking for serial Tpetra test suite.

significant (a factor of 3.1 for basic debug-mode runtime checking) it is still quite reasonable. When node tracing is enabled, the cost more than doubles to a factor of 6.8 times the basic optimized build. While the cost of full debug-mode runtime checking with node tracing is a factor of 6.8 over the basic optimized build, the cost of running with Valgrind is a factor of over 300! The increased cost of running Valgrind is a factor of 431.5 for the basic debug-mode executables. A factor of 300 can make running a tool like Valgrind prohibitive for even moderate sized problems while a factor of 6.8 may be quite reasonable. For example, a test problem that takes 20 minutes to run in a standard optimized build may take 2 hours 15 minutes to run with full debug-mode runtime checking with node tracing enabled but that same program may take 100 hours (i.e. more than 4 days) to run with Valgrind! Also, as has been mentioned several times before, in some respects the level of runtime checking provided by Teuchos in a debug-mode build is more effective that what one gets with just Valgrind[14]. In order to perform the most detailed runtime checking possible, one can run with Valgrind with debug-mode runtime checking with node tracing enabled. However, the overhead of this maximal checking is staggering at more than 23,000 times the cost of the basic optimized build! With this level of overhead, only very small test problems can be run.

What these timing results suggest is that the cost of debug-mode runtime checking for programs using the Teuchos memory management classes will be less than a factor of 10 more than the basic optimized build in the worst case while the overhead of running a tool like Valgrind can be as much as a factor of 400 or more. This means that enabling debug-mode runtime checking in regular development and automated testing is quite reasonable. Note that the Tpetra package used in this example is likely an extreme case in the usage of the Teuchos memory management classes. Other types of software that don't use the Teuchos memory management classes for such low-level computations will see much less of a slow-down. However, note that theses tests were only performed on one machine using one compiler so results on other platforms using different compilers may vary significantly.

### 5.11.6 Limitations of debug-mode runtime checking

Once memory is dynamically allocated and owned by one of the Teuchos memory management class objects, the debug-mode runtime checking will catch every imaginable type of programming error as long as a raw C++ pointer or raw C++ reference is not exposed. If all the idioms and rules outlined in this paper are followed, then the only issue the developer will have to address that is not 100% obvious are circular references. However, if programmers never made any mistakes, there would be no need for debug-mode

[14]However, Valgrind does perform a number of other types of checks including usage of uninitialized memory that are very useful and cannot be duplicated by the Teuchos memory management classes.

runtime testing in the first place. While the level of debug-mode runtime testing implemented in the Teuchos memory management classes is unmatched, code that converts from raw pointers (and raw references) to Teuchos memory management objects and vice versa is vulnerable to programming errors that the debug-mode runtime checking cannot catch.

The first category of programming errors that cannot be detected involve some types of conversions of raw pointers (and raw references) to Teuchos memory management objects. However, before discussing situations where the debug-mode runtime checking will not catch errors, first note that if an object is dynamically allocated and is immediately given over to a strong owning `RCP` object (or an `ArrayRCP` object in the case of arrays) then many different types of bad conversions from raw pointers (and raw references) to memory management types will be caught. That is because when an object's address is associated with a strong owning RCP, it gets added to the debug-mode `RCPNode` tracing system discussed in Section 5.9.1. Given this tracking, future conversions from a raw pointer or raw reference to a Teuchos memory management class object that result in multiple owning `RCP`s or dangling references from `Ptr`s and non-owning `RCP`s will all be detected and cleanly reported (see Sections 5.11.3 and 5.11.4). One way to guarantee this is to require that a classes' objects be dynamically allocated through its non-member constructors (Section 5.8.1) which returned the new objects wrapped in strong owning `RCP`s. In this way, the object is immediately tracked under the debug-mode node tracing system.

However, not every class can or should employ the non-member constructor idiom to force the creation of strong owning `RCP` objects. In particular, value-type classes (Section 4.1) such as `std::vector` and `Teuchos::Array` must be allowed to be generally constructed on the stack or globally but one still needs to be able to dynamically allocate them in many different situations. The downside to allowing value-type class objects to be dynamically allocated and managed with `RCP` is that it allows client code to try to create an owning `RCP` to a stack (or otherwise non-dynamically) allocated object which the debug-mode runtime checking will not be able to detect as shown, for example, in Listing 95.

**Listing 95** *: Example where debug-mode checking cannot detect an erroneous delete issue*

```
{
   std::vector<int> vec(n);
   const RCP<std::vector<int> > vec_rcp(&vec); // Gives ownership to delete!
   ...
   // When vec_rcp is destroyed it will call delete on the address &vec
   // resulting in undefined behavior (e.g.\ segfault)!
}
```

In this case, the owing `RCP<std::vector<int> >` object will try to call `delete` on the address `&vec` at the end of the block which will result in undefined behavior (e.g. segfault). The lack of debug-mode checking shown in Listing 95 is unfortunate but it is very hard to detect if an address is for a dynamically allocated object where it is okay to call `delete`[15]. Note that the code in this example violates Commandment 4 in Appendix B that states that owning `RCP` (and `ArrayRCP`) objects should only be created by passing in the address directly returned from `new` (or `new[]` for `ArrayRCP`) unless a customized deallocation policy object is attached which defines a more specialized dellocatioin policy. The good news though is that memory checking tools like Valgrind and Purify usually do a good job of detecting and reporting erroneous calls to

[15]Perhaps in the future a portable library function can be written and used that will be able to detect the difference between a stack address and a heap address so an exception can be thrown right when the bad owning `RCP` is first created.

`delete` (i.e. `free(...)`) that try to free stack-owned memory. But again if the idioms outlined in Section 5.8 and the commandments defined in Appendix B are followed, this problem should never occur.

The other category of programming errors that the debug-mode runtime checking cannot detect and report involves exposing and then misusing raw C++ pointers and references. As soon as client code exposes a raw C++ pointer and starts copying it around, all bets are off. However, even if client code never exposes a C++ pointer, one can still get into trouble. One unfortunate case involves the use of raw C++ references. If raw C++ references are only used as formal arguments to C++ functions, one will almost never have a problem. However, incorrectly returning an `RCP` object by reference instead of by value, as is described in Section 5.8.5, can result in invalid C++ references. Also, if one uses references like in Listing 96, then one can of course have dangling raw C++ references that the Teuchos debug-mode runtime checking can never catch.

**Listing 96** *: Example of where holding on to a raw C++ references disables debug-mode runtime checking*

```
RCP<A> a_ptr = newA();
A &a = *a_ptr;
...
a->someFunc();
// This above object may not be valid anymore and may result in
// undefined behavior (a segfault)!
```

The code in Listing 96 violates the use of raw C++ references only for non-persisting associations. The statement `A &a = *a_ptr` results in the creation of a persisting relationship in that it extends past the statement where it was created.

In summary, as soon as an object reference is exposed through a raw C++ pointer or a raw C++ reference, in general the Teuchos debug-mode runtime checking can no longer detect errors. Therefore, never expose a raw C++ pointer (except for the situations described in Section 5.2) and only expose and use raw C++ references for strictly non-persisting associations. Also, great care must be taken in first constructing Teuchos memory management class objects such they have the correct memory management properties.

### 5.11.7 Exception handling and debugging

The debug-mode runtime checking performed by the Teuchos memory management classes throw exceptions when violations are detected. As has been shown throughout this document, these exceptions have associated messages that contain detailed information about the nature and context of the problem.

All exceptions thrown by the Teuchos memory management classes (and the rest of Trilinos for that matter) all use a system of macros in the file `Teuchos_TestForException.hpp`. All of these macros call the function `TestForException_break(...)` just before an exception is thrown. Therefore, if the error is repeatable (and most errors are), then one can open a debugger (e.g. GDB) and set a break-point in that function, run the program, and then examine the state of the program just as the exception is being thorwn. Several exceptions can be thrown before the exception that one needs to debug. To make it easier to break on the exception that one cares about, every exception message has a `Thrown number` associated with it embedded in the error message of the exception object. One can set a conditional break-point in

`TestForException_break(...)` to only stop when `throwNumber` has the right value. For example, if one needs to stop on `Throw number = 10`, then in GDB one can set:

```
(gdb) b 'Teuchos::TestForException_break( [TAB] [ENTER]
(gdb) cond 1 throwNumber==10
(gdb) run
```

When the program stops at this break-point, one can then examine the call stack to troubleshoot the problem.

Many exception messages contain other types of information that would have one set breakpoints in other functions. For example, a dangling reference exception (as shown in Section 5.11.3) would contain addresses of objects that one would use to set conditional breakpoints. To examine the context under which an `RCPNode` is first created, one would set a break-point in the function `Teuchos::RCPNodeTracer::addNewRCPNode(...)` and set a condition to only break when `insertionNumber` is the number printed in the exception message. For example, for the exception message shown in Listing 88, one would set the break-point in GDB as:

```
(gdb) b 'Teuchos::RCPNodeTracer::addNewRCPNode [TAB] [ENTER]
(gdb) cond 1 insertionNumber==5
(gdb) run
```

When the debugger breaks, one would then be able to examine the call stack to see the context under which this `RCPNode` object is first created.

NOTE: Setting breakpoints based on `insertionNumber` is generally better than trying to set breakpoints based on the object addresses because the same address can get reused multiple times as objects are created and destroyed. Only the `insertionNumber` uniquely identifies a particular `RCPNode` object. In builds where node tracing is not enabled, `insertionNumber` will be equal to -1 and will not aid in debugging.

NOTE: Before entering a conditional break-point involving an address, one must first run the program again in the debugger which will typically produce an exception message with different object addresses because the debugger moves things around in memory. One will need to use these new pointer addresses when setting conditional breakpoints.

The Teuchos reference-counting classes are all fully exception safe in that they provide either the basic guarantee (retain some valid object state and no leaked memory when an exception is thrown), the strong guarantee (retain original state when an exception is thrown), or the no-throw guarantee (see [31, Item 71]). However, if exceptions are thrown from destructors when objects are being destroyed, then the reference-counting classes are only fully exception safe in a debug-mode build. This does not really break exception safety since destructors should not be throwing exceptions in most valid C++ programs (see [31, Item 51]). The Teuchos memory management classes provide the foundation for allowing the wide-spread and consistent use of C++ exception handling in all client code in such a way as memory will not be leaked when exceptions are thrown. However, achieving a truly exception safe program means more than just not leaking memory; it means that all code provides at least one of the fundamental exception guarantees (again, see [31, Item 71]).

Note that throwing exceptions differs from what many other class libraries do which is typically to call `assert(...)` when a runtime failure is discovered. For example, the checked STL for g++ will call assert

**GCC 4.1.2:** GNU GCC 4.1.2 (compiler options `-O3 -DBOOST_SP_DISABLE_THREADS`) running under Linux 2.6.18-128.1.6.el5 on 2 Quad Intel Xeon CPUs at 2.93GHz and 4MB L1 Cache and 16 GB RAM.

**ICC 10.1:** Intel ICC C++ 10.1 (compiler options `-O3 -DBOOST_SP_DISABLE_THREADS`) running under Linux 2.6.18-128.1.6.el5 on 2 Quad Intel Xeon CPUs at 2.93GHz and 4MB L1 Cache and 16 GB RAM.

**MSVC++ 2008:** Microsoft Visual C++ 2008 (compiler options `/D_SECURE_SCL=0 /DBOOST_SP_DISABLE_THREADS /Ox`) running under Windows Vista Enterprise on an Intel Core 2 Duo CPU T9800 at 2.93GHz and 2.00 MB RAM.

**Table 17.** Performance testing platforms.

when a usage violation is discovered. There are pros and cons for throwing exceptions versus halting the program but if code can be made exception safe, then one can argue that throwing exceptions is better because it allows the program to recover in case of a catastrophic failure of a submodule while calling `assert(...)` does not. Also, writing unit tests for code that throws exceptions is much easier and more efficient than trying to write unit tests for code that halts the program. This issue of testability is a huge advantage of exception handling over calling `assert(...)` or `exit(...)` when an error occurs.

## 5.12 Optimized performance

While debug-mode runtime checking is of great importance, of equal importance is speed in a optimized non-debug build. It is critical in high performance code that the wise use of the Teuchos memory management classes lead to optimized performance that is nearly identical to the performance of raw pointers. Otherwise, if there is always a performance gap with using the Teuchos memory management classes, then there will always be an excuse to go back to using raw pointers will all of the disastrous consequences discussed in Section 1 and Section 2.

In this section, the optimized performance of the Teuchos memory management classes is analyzed. In an optimized build, all of the runtime checking is disabled but there is still some non-trivial overhead associated with the reference-counting machinery. If used at too fine a granularity, reference-counting overhead can become a significant space/time performance problem on real-world problems.

The optimized performance of several different types of operations are examined in the next few sections. All of these performance timing tests were run on three different compilers shown in Table 17 that represent two mainstream platforms. The GCC 4.1.2 and Intel ICC 10.1 results where run on the same Linux machine and therefore one can directly compare the optimizing capability of these two compilers on this platform. Note that the processor used for the Microsoft Vista platform is also Intel and has the same clock speed as for the Linux platform. Therefore, one can make fairly direct comparisons of runtimes between the three different compilers. Timings on other compilers may give different results, especially for compilers that have a bad history at optimizing C++ code (e.g. PGI, Sun, AIX etc.). All of these performance timing tests are driven by a performance testing framework in Teuchos and there are nightly performance tests that strictly enforce relative performance timing targets.

This section is broken up into subsections as follows. First, the optimized performance of the reference-counting machinery is looked at in Section 5.12.1. Reference-counting overhead will never go to zero with respect to raw pointers but it is constant-time overhead and therefore its impact can be minimized by not applying it at too low a level. The optimized performance of the Teuchos array classes is given in Section 5.12.2. The timing results show that the basic bracket operator (i.e. `a[i]`) and iterator (i.e. `a.begin()`) access methods all yield raw pointer performance. Finally, in Section 5.12.3, performance tuning strategies are discussed primarily addressing the issue of performance optimizations related to semi-persisting associations.

### 5.12.1 Reference counting overhead

While the reference-counting machinery used by the `RCP` and `ArrayRCP` classes significantly improves software development productivity and quality in many respects, it also has a certain amount of space and time overhead that needs to be considered in design decisions. Here, the cost of the various operations associated with the `RCP` class are compared to raw pointers and to the `boost::shared_ptr` class. Timings are performed for creating and destroying the `RCPNode` object and reference-counted object, for manipulating the reference count, and for accessing the underlying reference-counted object. These are the core operations of the `RCP` class that are most likely to affect performance.

All of the operations being timed are very low-level and therefore it is difficult to get meaningful unbiased timing results. To get accurate timings, one must perform the operation in loop and average the times. With naive code, some compilers (e.g. Microsoft Visual C++) will just optimize away the entire loop. Therefore, the operation must be performed in the context of a loop over an array where the result of the loop gets used in some way to accumulate a final result. Examples of these types of timing loops will be given below. Because of the loop and iterator overhead and this extra (minimal) computation, the timings listed for each operation are higher that what they would be otherwise. Therefore, the overhead reported is lower that what it really is but by how much one cannot be sure. Also, when performing loops, issues of loop initialization and cache issues come into play. In order to avoid these issues, a single loop size from all the results of 1024 was selected to display in the figures and tables in this section. The raw timing results for other loops sizes are given in Appendix D.1.

Note that the atomic thread-safe reference-counting machinary in `boost::shared_ptr` was turned off in order to get better timing comparisons. Preliminary timing studies showed that the assembler-optimized atomic lock-free reference-counting machinary on Linux/GCC imparted about a 4x overhead. To avoid this performance overhead, the assember code for atomic reference-count manipulation was disabled by compiling with `-DBOOST_SP_DISABLE_THREADS`. Issues of thead safety are briefly discussed in Section 5.14.

The first type of overhead to consider is the memory overhead of the reference-counting machinery shown in Figure 4. Table 18 shows the sizes of some important objects associated with `RCP` and `boost::shared_ptr` (on a 64 bit platform where pointers are 8 bytes). The sizes are shown for allocating `std::vector<double>` objects but the memory used by the reference counting machinery only depends on pointers so the memory usage overhead is the same no mater what type of object is used. From looking at Table 18, one can see that the static size of `std::vector<double>` is 24 bytes for this compiler. Consider allocating an `std::vector<double>` object with only one element. This would dynamically allocate one `double` object in an array giving a total of 32 bytes. Now consider the reference-counting machinery overhead. For every allocated `std::vector<double>` object, there is a reference-counting

| Type | sizeof(Type) |
|---|---|
| bool | 1 |
| double* | 8 |
| double | 8 |
| std::vector<double> | 24 |
| boost::shared_ptr<std::vector<double> > | 16 |
| boost::detail::sp_counted_impl_p<std::vector<double> > | 32 |
| RCP<std::vector<double> > | 24 |
| RCPNodeImpl<std::vector<double>, ... > | 48 |

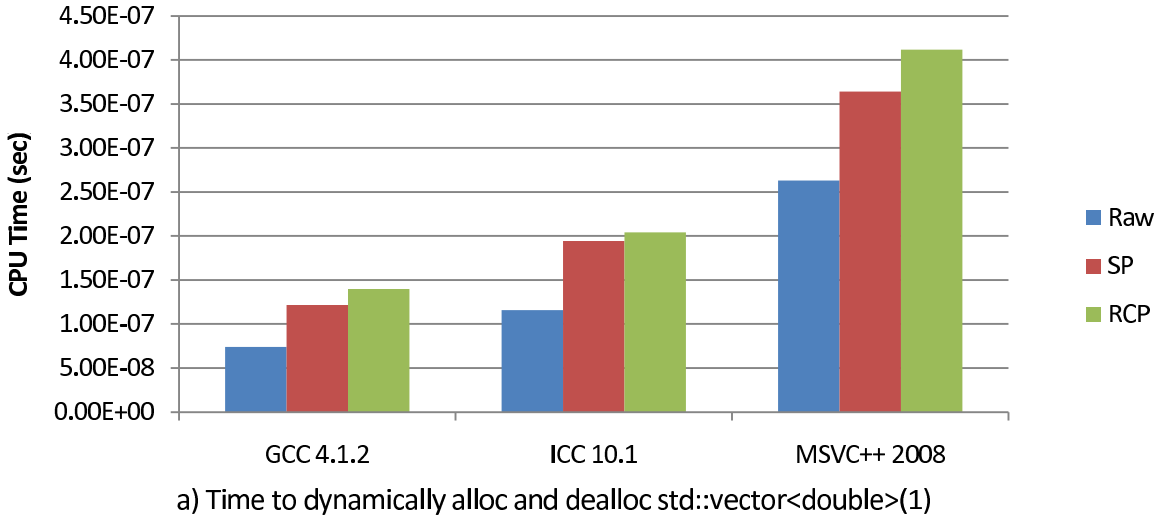
**Table 18.** Sizes of RCP and boost::shared_ptr objects for 64 bit GCC 4.1.2.

node object of type RCPNodeImpl<std::vector<double>, ... > which is 48 bytes. In addition there is also an RCP<std::vector<double> > object of size 24 bytes. That gives a total of 24+48=72 bytes of reference-counting overhead to manage an object that only consumes 32 bytes. That is memory overhead of 225%! However, when the std::vector<double> is allocated to hold 100 elements, the memory consumed by the std::vector<double> object is 24+8*(100) = 824 bytes. Now the 72 bytes of reference-counting overhead is only 8.7%. By the time one gets to 1000 elements, the overhead drops to 0.8%. The point is that the reference-counting machinery imparts a storage overhead that is non-trivial for small objects. Therefore, RCP should not be used to manage large numbers small objects. Likewise, ArrayRCP should not be used to manage large numbers of small arrays for the same reason.

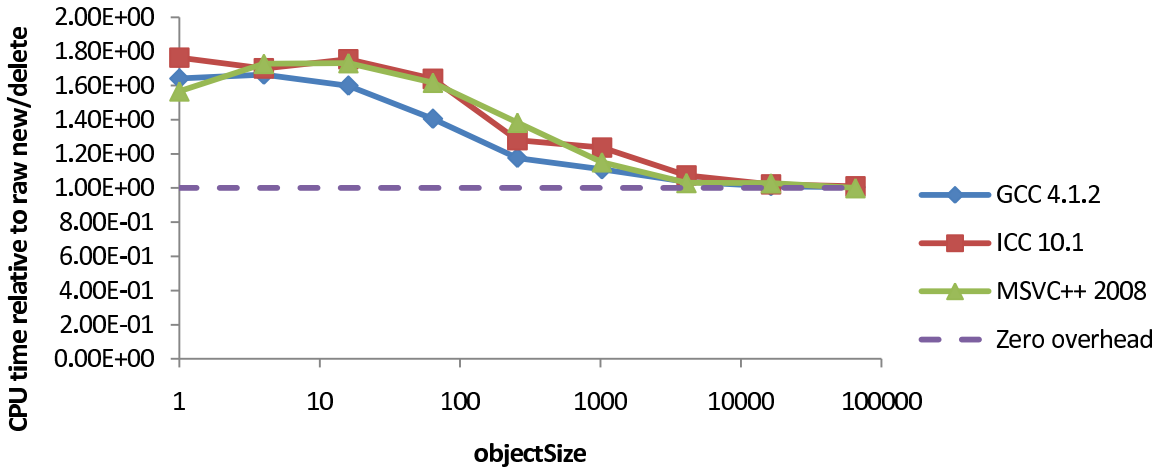
Table 18 also shows the sizes of comparable objects associated with the boost::shared_ptr class. The boost sp_counted_impl_p node object only consumes 32 bytes on this machine as apposed to the 48 bytes for the RCPNodeImpl object. The increased overhead of the RCPNodeImpl object is due to the pointer for the extra data map, an extra ownership Boolean, and storage of the deallocator object. Also, the boost::shared_ptr object itself only consumes 16 bytes while the equivalent RCP object uses 24 bytes. This increase in storage is due to having to store a strength enum to dynamically handle STRONG and WEAK references. This is the storage cost of increase flexibility of the RCP class over the boost::shared_ptr class.

Now consider the runtime overhead associated with dynamic allocation and deallocation. Figure 12 shows the timings for dynamically allocating and deleting std::vector<double> objects for different numbers of vector elements on the three compilers shown in Table 17. Figure 12.a shows the timings for allocating std::vector<double> objects with only one element. This shows that there is some runtime overhead needed to dynamically allocate new node objects for RCP. The extra overhead is due to an extra call to new in order to allocate the node object. Note that the extra overhead for RCP is quite small with respect to boost::shared_ptr for all three compilers (because both classes do very similar things). However, this is constant time overhead so as larger std::vector<double> objects are allocated (with associated initialization of the vector elements in an inner loop) the relative overhead goes to zero, as shown in Figure 12.b. Therefore, the runtime overhead of the reference-counting machinery for allocating and deallocating large objects is very small.

Now consider timings for dereferencing using RCP::operator*(), member access through the arrow operator RCP::operator->(), and assignment through RCP::operator=(...) (which changes the




a) Time to dynamically alloc and dealloc std::vector<double>(1)




b) Time to dynamically alloc and dealloc std::vector<double>(objectSize) using RCP

**Figure 12.** Timings for allocating and deallocating objects using RCP

reference counts) shown in Figure 13. These timings are the average CPU time (in seconds) per inner loop iteration (see Listing 97 for an example). These timing results show that dereferencing and member access for `RCP` yield raw pointer performance on all the compilers because these member functions are trivially inlined to expose the raw pointer.

The assignment operator, however, imposes significant overhead because of the need to increment and deincrement the reference counts. The timing code fragment that exercises `RCP::operator=(...)` is shown in Listing 97. (Note that `std::vector` is used instead of `Array` in Listing 97 in order to avoid timing overhead that might result from a bad implementation of `Array::operator[](...)` that would affect the timing results.)

**Listing 97** *: Performance timing loops for `RCP::operator=(...)`*

```
{
  RCP<char> p(new char('n'));
  std::vector<RCP<char> > p_vec(arraySize);
  TEUCHOS_START_PERF_OUTPUT_TIMER_INNERLOOP(outputter, numActualLoops, arraySize)
  {
    for (int i=0; i < arraySize; ++i) {
      p_vec[i] = p;
      // NOTE: This assignment operation tests the copy constructor and
      // the swap function.  This calls both bind() and unbind()
      // underneath.
    }
  }
}
TEUCHOS_END_PERF_OUTPUT_TIMER(outputter, rcpTime);
```

Timing results for the code in Listing 97 for `numActualLoops=338498` and `arraySize=1024` are shown in Figure 13 along with similar timings for raw pointers and `boost::shared_ptr`. The full timing results for other sizes are show in in Appendix D.1.

Several interesting points to note about these timing results are described below.

First, the timing results for the simple raw-pointer loops shown in Figure 13 suggest that these two machines have nearly identical processor speeds. Therefore, the CPU times on the Y-axis scale for each of these compiler/machine bar charts is made the same to allow for absolute comparisons. This allows for direct comparisons of the optimizing capabilities of these three compilers with respect to dealing with general C++ code (and not just C-like raw pointer loops). This suggests that GCC 4.1.2 is better than the rest and that MSVC++ 2008 is quite bad at optimizing general `RCP` C++ code.

Second, note that the cost of manipulating the reference count in `RCP::operator=(...)` is an order of magnitude higher than the dereference and arrow operators which have raw-pointer performance. The real overhead of manipulating the reference counts may not actually be this high due to the simple nature of the raw pointer code run in a loop getting better optimization. The reference-count manipulation code involves if statements that may disable certain loop optimizations.

Third, note that `RCP::operator=(...)` is about 30% slower on GCC 4.1.2 than for `boost::shared_ptr` due to the extra overhead of dynamically handling strong and weak reference counts. The overhead of `RCP` over `boost::shared_ptr` goes up to 50% on on ICC 10.1 and then falls off a cliff going up to 300% for

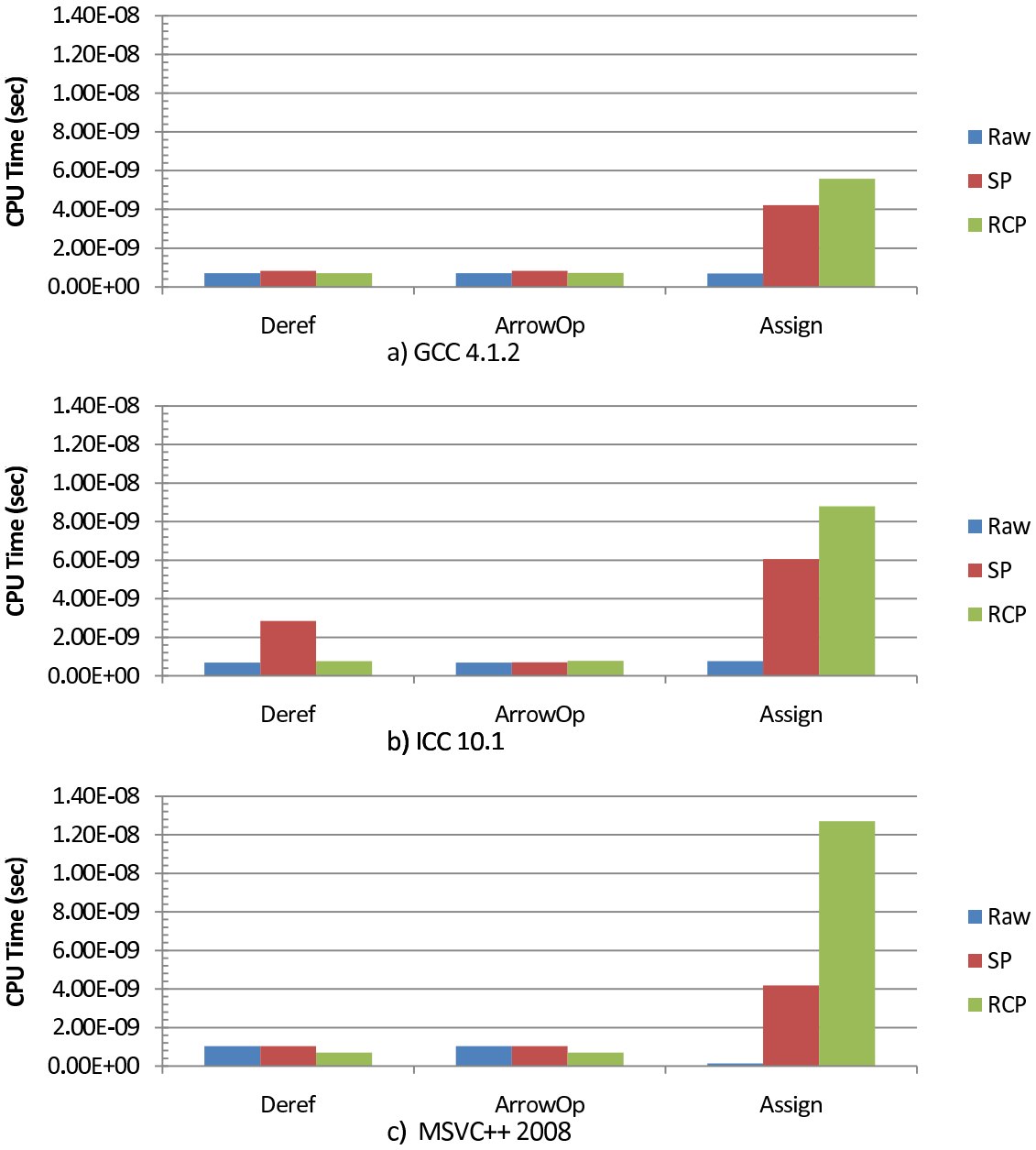



**Figure 13.** Timings of basic RCP operations on for three compilers

MSVC++ 2008. Clearly the MSVC++ compiler is not inlining the `RCP` functions as well in this case. However, there may be compiler options that would cause the MSVC++ compiler to be more aggressive in inlining but none could be found after a moderate level of experimentation.

Fourth, note that for GCC 4.1.2, the cost of manipulating the reference count (at `5.59-09 sec`) is two orders of magnitude less than the cost to allocate and deallocate an `std::vector<double>` object with only one element (at `1.39-07 sec`) and is three orders of magnitude less for 16384 elements (at `5.84-06 sec`) as shown in Appendix D.1. Therefore, just the memory allocation overhead can dominate these other costs in some cases. Also, if a large object is being used with expensive operations, then the reference-counting overhead will be insignificant compared to using the object. Again, this argues that classes like `RCP` should only be used to manage larger objects that have more expensive operations associated with them. The same argument can be made that `ArrayRCP` should only be used for managing larger arrays of data where the cost of loops over the data overwhelm the reference-counting costs.

Lastly, note that the `RCP::operator=(...)` implementation both deincrements and increments the reference count while the copy constructor only has to increment the reference count. Therefore, we might expect that the copy constructor would be about twice as fast as the assignment operator. The performance of the copy constructor is not measured in a loop because it is hard write a loop that tests it without other overhead. However, the fastest approach is to avoid the copy of the `RCP` objects at all by passing in constant references to the `RCP` objects as formal function arguments which is advocated in Section 5.8.4.

### 5.12.2 Array access and iterator overhead

Another important type of performance (perhaps more important than the performance of `RCP` for handling single objects) is the performance of the Teuchos array classes. In an optimized non-debug build these classes must yield the same performance as using raw pointers or the performance of the application will definitely be affected.

Performance timing experiments for the bracket operator `operator[](size_type)` and iterators (returned from the `begin()` and `end()` functions) were performed using simple timing loops. Unlike the performance tests for `RCP` described in the previous section, timing array operations naturally lends themselves to performance timings. The performance timing code fragments for the `Array` class are shown in Listing 98 and 99. The timing loop code for raw pointers and the `ArrayRCP` and `ArrayView` classes are nearly identical.

**Listing 98** *: Performance timing loops for `Array::operator[](size_type)`*

```
Teuchos::Array<double> a(arraySize);
TEUCHOS_START_PERF_OUTPUT_TIMER_INNERLOOP(outputter, numActualLoops, arraySize)
{
  for (Ordinal i=0; i < arraySize; ++i)
    a[i] = 0.0;
}
TEUCHOS_END_PERF_OUTPUT_TIMER(outputter, arrayTime);
```

**Listing 99** *: Performance timing loops for `Array` iterators*

```
Teuchos::Array<double> a(arraySize);
```

```
TEUCHOS_START_PERF_OUTPUT_TIMER_INNERLOOP(outputter, numActualLoops, arraySize)
{
  Teuchos::Array<double>::iterator a_itr = a.begin(), a_end = a.end();
  for ( ; a_itr < a_end; ++a_itr)
    *a_itr = 0.0;
}
TEUCHOS_END_PERF_OUTPUT_TIMER(outputter, arrayTime);
```

Figure 14 shows the CPU timings for `operator[](size_type)` and iterators for the performance tests with sizes `numActualLoops=230574` and `arraySize=1600`. These timing results are fairly interesting and there are a few important details to note.

First, the performance of the raw pointer iterator loops on all three platforms is almost identical. The CPU time per inner loop iteration for the raw iterator form of the loop for all three compilers on the two different Linux and Windows machines is about `3.5e-10` seconds. This suggests that the CPUs on these two machines are nearly identical for low level operations. This therefore suggests that only the compilers that result in different performance for the other operations. Because these processors appear to be giving the same raw performance, the Y-axis scales on all the timing bar charts are made the same in Figure 14 to allow for direct comparison between each of the platforms.

Second, the ICC 4.1.2 compiler does not optimize the GNU `std::vector::iterator` type[16] as well as it optimizes raw pointer iterator syntax. The GNU GCC 4.1.2 compiler itself does not even quite fully optimize its own `std::vector::iterator` type!

Third, the ICC 4.12 compiler is not optimizing the array indexing form of the inner loops shown in Listing 98 as well as the other two compilers even for the use of raw pointers. The performance is even worse for the abstract data types `std::vector`, `Array`, `ArrayRCP`, and `ArrayView`.

Forth, for some reason the MSVC++ 2008 compiler is not optimizing the loop using `ArrayRCP::operator[](size_type)` as well as for the other array types. Several different compiler options and variations on the `ArrayRCP` and performance timing code where experimented with without any impact (except to make every operation slower). However, the performance timing loop using iterator for `ArrayRCP` yield raw-pointer performance (not surprising given that `ArrayRCP::iterator` is just a raw pointer in a non-debug build).

The take-away points from these timing results are that all compilers do not fully optimize the array indexing form of the loops and only raw pointers used as iterators will yield the optimal performance. The Teuchos array classes perform at least as well as `std::vector` and actually out-perform `std::vector` in some cases. This is only because the Teuchos array classes use raw C++ pointers for iterators in an optimized non-debug build while the GNU implementation `std::vector` uses a library-defined class.

### 5.12.3 Performance tuning strategies, semi-persisting associations

The timing results shown in the prior two sections lead to a few different conclusions related to performance issues in the use of the Teuchos memory management types:

[16] The data-type for the optimized iterator for `std::vector::iterator` on GNU is not a raw pointer. Instead it is a library-defined data type with all inline functions that should in theory be optimized as well as the raw pointer but not always.

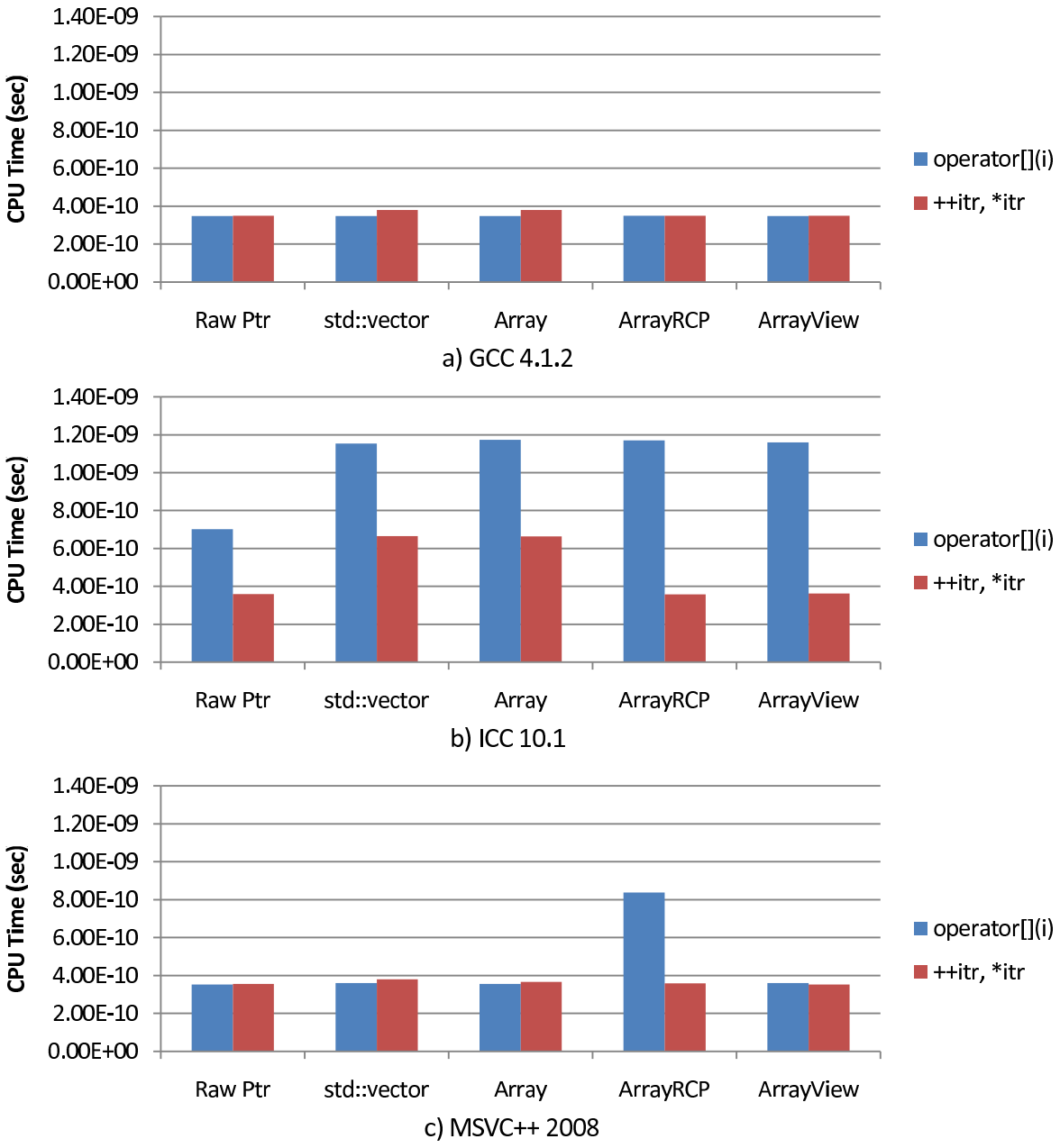



**Figure 14.** Timings for basic Array, ArrayRCP, and ArrayView operations

- The reference-counting machinery of the `RCP` class imparts at least 72 bytes of overhead for every reference-counted object (the overhead for `ArrayRCP` is slightly higher) and therefore `RCP` should not be used to manage massive numbers of small objects, just from a memory usage standpoint. Likewise, `ArrayRCP` should not be used to manage large collections of small arrays since the memory overhead could be significant.
- The extra runtime overhead of reference-counting machinery does not significantly increase dynamic memory allocation and deallocation runtime costs for moderately large objects.
- The runtime reference-counting overhead of `RCP` with respect to `boost::shared_ptr` can vary from as little as 30% on a good optimizing compiler (e.g. GCC 4.1.2) to as much as 300% or more on a poor optimizing compiler (e.g. MSVC++ 2008). Therefore, if portable performance of is critical, make sure and use the reference-counting types at as high a level of granularity as one can such that it does not damage the quality of the software (i.e. safety, flexibility, usability, maintainability).
- The most portable way to achieve high performance in array operations is to use iterators on `ArrayRCP` and `ArrayView` objects. The timing results on ICC 10.1 show that some compilers will not even given optimize performance for `std::vector::iterator`! Also, all compilers do not automatically fully optimize the array bracket form of an array-based loop so an iterator loop is the only foolproof way to get the best optimized performance across platforms.

The performance tests described above show that the memory and runtime overhead of the reference-counting machinery can be high when used with small cheap objects. Therefore, if the reference-counted types `RCP` and `ArrayRCP` are used at too low a level of granularity, the overall performance of the program may suffer and may use significantly more memory than it would with raw pointers. In such low-level code, strictly adhering to the idioms described in Sections 5.8.4 and 5.8.5 with respect to persisting relationships can significantly degrade performance. Therefore, in low-level performance-critical code, the strict idioms related to persisting associations need to be relaxed or the design must be changed to raise the level of granularity where the reference-counted types are used.

However, just because one cannot use `RCP` and `ArrayRCP` in every situration and still achieve high performance does not mean that the code should hard-code the use ofraw pointers. Instead, `Ptr` can be used instead of `RCP` and `ArrayView` can be used instead of `ArrayRCP` when the full semantics of persisting relationships are not needed and instead only semi-persisting relationships are needed (see Section 4.2). By using the types `Ptr` and `ArrayView` instead of raw pointers for all semi-persisting associations one still gets all of the strong debug-mode runtime checking that is described in Section 5.11 (e.g. dangling reference checking, null checking, range checking, etc.) yet these types remove all overhead over raw pointers in a non-debug optimized build.

As an example, consider the design of a sparse matrix class that stores its data as compressed sparse rows and allows access to the sparse rows. The client of the sparse matrix class would obtain handles to the sparse row data, make modifications to it, and then release the handles. Performing all of these operations in a single statement (as is strictly required for a non-persisting relationship as defined in Section 4.2) is impractical. Therefore, the handles for the internal sparse row data represent a persisting relationship and the strict interpretation of the idioms defined in Sections 5.8.4 and 5.8.5 would require the use of `ArrayRCP` yielding the sparse matrix class interface shown in Listing 100.

**Listing 100** *: Sparse matrix class interface adhering to strict interpretation of idioms for persisting relationships*

```
class SparseMatrix {
public:
  ...
  int getNumRows() const;
  void getSparseRow( int rowId, const Ptr<ArrayRCP<double> > &values,
    const Ptr<ArrayRCP<const int> > &colIds);
  ...
};
```

The `SparseMatrix` class shown in Listing 100 would be used as shown in Listing 101.

**Listing 101** *: Client code using `ArrayRCP` form of the `SparseMatrix` class*

```
void zeroOutSparseMatrix(const Ptr<SparseMatrix> &M)
{
  const int numRows = M->getNumRows();
  for (int row_i = 0; row_i < numRows; ++row_i) {
    ArrayRCP<double> values;
    M->getSparseRow(row_i, outArg(values), null);
    typedef ArrayRCP<double>::iterator itr_t;
    for (itr_t itr = values.begin(); itr != values.end(); ++itr)
      *itr = 0.0;
  }
}
```

While the interface and the user code shown in Listing 101 strictly satisfies that safe and bullet-proof idioms on persisting associations described in Section 5.8.4, the reference counting overhead (in memory size and speed) of this code can be quite high if the rows are very sparse. Looking at code such as shown in Listing 101 and similar use cases, it never seems reasonable that a client would grab `ArrayRCP` objects to internal rows and expect to have the row data persist even if the matrix changed structure or was deleted. Instead, one just needs to set up the infrastructure for semi-persisting associations to be able to detect those types of invalid usage in a debug-mode build but yield high performance in an optimized build. Therefore, it seems reasonable to replace `ArrayRCP` in `SparseMatrix::getSparseRow(...)` in Listing 100 with `ArrayView` yielding the new `SparseMatrix` interface shown in Listing 102.

**Listing 102** *: Sparse matrix class interface using a semi-persisting association for row views for the sake of performance*

```
class SparseMatrix {
public:
  ...
  int getNumRows() const;
  void getSparseRow( int rowId, const Ptr<ArrayView<double> > &values,
    const Ptr<ArrayView<const int> > &colIds);
  ...
};
```

The updated client code zeroing out the rows of the matrix would then look like Listing 103.

**Listing 103** *: Client code using `ArrayView` form of the `SparseMatrix` class with semi-persisting row views*

```
void zeroOutSparseMatrix(const Ptr<SparseMatrix> &M)
{
  const int numRows = M->getNumRows();
  for (int row_i = 0; row_i < numRows; ++row_i) {
    ArrayView<double> values;
    M->getSparseRow(row_i, outArg(values), null);
    typedef ArrayView<double>::iterator itr_t;
    for (itr_t itr = values.begin(); itr != values.end(); ++itr)
      *itr = 0.0;
  }
}
```

Now the client code in Listing 103 will have no reference-counting overhead in a non-debug optimized build but in a debug-mode build, all invalid usage will be detected. For example, consider invalid code such as shown Listing 104 where the client code tries to hold on to sparse row data after the matrix is deleted.

**Listing 104** *: Example of invalid usage of `SparseMatrix` leading to a dangling reference exception in a debug-mode build*

```
// Create and initialize the matrix
RCP<SparseMatrix> M = createSparseMatrix(...); // Non-member constructor
...

// Grab a sparse row to the matrix
ArrayView<double> values_row_0;
ArrayView<const int> colIds_row_0;
M->getSpaseRow(0, outArg(values_row_0), outArg(colIds_row_0));

// Delete the matrix (leaving dangling values_row_0 and colIds_row_0)
M = null;

// Try to access the row
ArrayView<double>::iterator
  itr = values.begin(),     // Throws exception in debug-mode build!
  itr_end = values.end();
for ( ; itr != itr_end; ++itr)
  *itr = 0.0;
```

As shown in Listing 104, using `ArrayView` allows programming errors to be detected in a debug-mode build. If raw pointers would have been used, this dangling reference may not be detected right away. On some platforms for some problem sizes, the program using raw pointers may seem to run just fine and Valgrind may not complain (especially if sophisticated memory management is used inside the `SparseMatrix` class). The error may not present itself until months or years later where it may do untold harm.

Note that there may be some extreme cases where the overhead of an extra size data member in `ArrayView` is too high. In these cases, one can instead use an iterator type such as `Array::iterator` or

`ArrayRCP::iterator` (depending on type of the underlying container class). In a debug-mode build, the iterator objects will be fully checked `ArrayRCP` objects while in a non-debug optimized build, the iterators will be raw pointers (or `std::vector::iterator` in the case of `Array::iterator`). This yields raw pointer performance in a non-debug optimized build with no space or time overhead (because all the objects actually are raw pointers in this case).

The point of this section is to acknowledge that there will be situations in low-level code where the strict adherence to using reference-counted types `RCP` and `ArrayRCP` for persisting associations may not yield acceptable performance and therefore one must instead provide for semi-persisting views. However, as demonstrated above, the solution to the performance problem is not to fall back to using raw pointers but instead to fall back on the non-reference-counted types `Ptr` and `ArrayView` (or `Array[RCP]::iterator`). By using the types `Ptr` and `ArrayView` (or `Array[RCP]::iterator`), one maintains all the desirable debug-mode runtime checking without any of the reference-counting overhead in a non-debug optimized build. We can have our cake and eat it too!

## 5.13 Related idioms and design patterns

There are a number of important idioms related to the usage of the Teuchos memory management classes and most specifically the `RCP` class. The power and flexibility of the reference-counting machinery built in to the `RCP` class opens the door the a whole host of interesting idioms, a few of which are described in the following subsections.

### 5.13.1 The inverted object ownership idiom

A rare situation that can occur is when one has an object that maintains an `RCP` to another object but one wants to expose the second object and have it remember the first object; in other words, one wants to invert the object ownership. To demonstrate, consider the two classes in Listing 105.

**Listing 105** *: Two classes where one maintains an RCP to the other*

```
class A { ... };

RCP<A> createA(...);

class B {
public:
  static RCP<B> create(const RCP<A> &a) {return rcp(new B(a)); }
  RCP<A> getA() { return a_; }
  void unsetA() { a_ = null; }
  ...
private:
  RCP<A> a_;
  B(const RCP<A> &a) a_(a) {}
};

RCP<B> createB(const RCP<A> &a) {return B::create(a);}
```

The class `A` in Listing 105 may involve some complex initialization or it may only be an abstract interface with multiple subclasses. In either case, it may make sense to provide a factory function (or a set of such functions) that creates and initializes a `B` object for different complex initializations of `A` objects such as the example shown in Listing 106.

**Listing 106** *: A factory function that creates a `B` object wrapping a complex `A` object*

```
RCP<B> createBFactory(...)
{
   // Complex initialization of A
   RCP<A> a;
   ...
   // Wrapped B
   return createB(a);
}
```

Up to now, this is pretty standard code. The client would typically hold an `RCP<B>` object and would manage the lifetime of the `A` object implicitly wrapped in the `B` object.

However, now consider a rare use case where a client may only want to deal directly with the `A` object but still maintain the `B` object for later use. There are a few approaches that one could try to implement this inversion of RCP ownership but there is a way to enable this that is 100% bullet-proof without having to change the existing `A` or `B` classes or any other code at all. The way to do this is to use the `rcpWithInvertedObjOwnership(...)` function (defined in Listing 108) as shown in Listing 107.

**Listing 107** *: A factory function that returns an `A` object embedded with a `B` object (inverting the ownership relationship)*

```
RCP<A> createAFactory(...)
{
  RCP<B> b = createBFactory(...);
  return rcpWithInvertedObjOwnership(b->getA(), b);
}
```

**Listing 108** *: Standard helper function implementing the "inverted object ownership" idiom*

```
template<class T, class ParentT>
RCP<T> rcpWithInvertedObjOwnership(const RCP<T> &child, const RCP<ParentT> &parent)
{
  typedef std::pair<RCP<T>, RCP<ParentT> > Pair_t;
  return rcpWithEmbeddedObj(child.getRawPtr(), Pair_t(child, parent), false);
}
```

Without going into a lot of detail, what the code in Listings 107 and 108 accomplishes is that it defines a new `RCP<A>` object with a new `RCPNode` object that uses the other existing `RCP<A>` and `RCP<B>` objects to define ownership and ensure that the underlying `A` and `B` objects do not go away until the last `RCP<A>` object copied from the object returned by the function `createAFactory(...)` has gone away. The reason that `false` is passed into the `rcpWithEmbeddedObj(...)` call is because it is the embedded objects `RCP<A>`

and RCP<B> that define the deallocation and not the embedded deallocator (which calls delete). The reason that both RCP<A> and RCP<B> are passed as an embedded object (stored in an std::pair object) is that one needs to make sure the A object does not get deleted in case some client calls the B::unsetA() function.

Given this data-structure, another piece of code can then extract the underlying RCP<B> object is shown in Listing 109 which uses the standard Teuchos function getInvertedObjOwnershipParent(...) defined in Listing 110.

**Listing 109** *: A function that extracts the B object from the A object*

```
RCP<B> extractBFromA(const RCP<A> &a)
{
  return getInvertedObjOwnershipParent<B>(a);
}
```

**Listing 110** *: Standard helper grabbing the inverted parent*

```
template<class ParentT, class T>
RCP<ParentT> getInvertedObjOwnershipParent(const RCP<T> &invertedChild)
{
  typedef std::pair<RCP<T>, RCP<ParentT> > Pair_t;
  Pair_t pair = getEmbeddedObj<T, Pair_t>(invertedChild);
  return pair.second;
}
```

That is all there is to it. This is not the sort of thing that one wants to expose to general clients but it can be very handy to have this type of flexibility when implementing the guts of library code. The above example shows the flexibility of these memory management classes and what some of the possibilities are if one understands the underlying reference-counting machinery a little.

### 5.13.2 The separate construction and just-in-time initialization idioms

The "separate construction and initialization" and "just-in-time initialization" idioms described here are not specific to the use of the Teuchos memory management classes but they do provide the basic foundation for the next idiom described, the "object self-reference" idiom. To set up the context for the discussion, consider a typical class design shown in Listing 111.

**Listing 111** *: Example of a typical C++ class that uses constructors for all initialization*

```
class SomeClass : public SomeBaseClass {
  int member1_;
  double member2_;
  RCP<A> a_;
  RCP<B> b_;
  RCP<C> c_;
  void finalInitialization() { ...} // Can't call virtual functions on SomeBaseClass
```

```
public:
  SomeClass(): member1_(1), member2_(5.0) {}
  SomeClass(const RCP<A> &a) : member1_(1), member2_(5.0), a_(a)
    { finalInitialization(); }
  SomeClass(const RCP<B> &b) : member1_(1), member2_(5.0), b_(b)
    { finalInitialization(); }
  SomeClass(const RCP<A> &a, const RCP<B> &b, const int someValue)
    : member1_(1), member2_(5.0), a_(a), b_(b)
    { c_ = createC(rcp(this, false), a, b, someValue);
      finalInitialization(); }
  RCP<A> get_A() { return a_; }
  RCP<B> get_B() { return b_; }
  RCP<C> get_C() { return c_; }
  void doSomeOperation(...) {...}
};
```

So what is wrong with the design of `SomeClass` in Listing 111? First, there is the duplication of default values for `member1_`, `member2_` in all of the constructors. This makes it labor intensive and error-prone to change the values later. Yes, one could create static constants of some type to be reused in all the constructor initialization lists but one still has to list all of these arguments in every constructor[17].

The second problem with the class `SomeClass` that it cannot call any virtual functions in the base class `SomeBaseClass` to help initialize its state in the constructors [31, Item 49].

The third problem with the design of `SomeClass` shown in Listing 111 is that the `C` object that is created in the third constructor that takes `A` and `B` objects is not properly setting up a persisting relationship between the `C` object and the `SomeClass` object. When this `C` object is exposed through the `get_C()` member function, this creates a dangerous situation where the `SomeClass` object may be deleted leaving a client with a dangling `RCP<C>` object with no way for the reference-counting machinery described in Section 5.9 do detect the the problem. This issue will be discussed more in the context of the "object self-reference" idiom in Section 5.13.3.

Lastly, the class `SomeClass` is inflexible in that it requires the client creating the `SomeClass` object to know the concrete types of `A` and or `B` (which could be abstract interfaces in this example) right when the `SomeClass` object is first created. This creates a three-way coupling between a) the client, with b) defining the time when the `SomeClass` object is first created, and c) needing fully constructed `A` and `B` objects right when `SomeClass` is first created. In complex programs, it is very hard and very constraining to have to fully initialize a web interconnected objects before constructing downstream objects.

Without further ado, the use of the "separate construction and initialization" and "just-in-time initialization" idioms applied to `SomeClass` shown in Listing 111 gives the new refactored class in Listing 112.

**Listing 112** *: Example of the use of the "separate construction and initialization" and "just-in-time initialization" idioms*

```
class SomeClass : public SomeBaseClase {
```

[17] The new C++0x standard will address the problem of duplicate constructor initialization lists by allowing constructors to call each other but we will not see such a feature in wide spread use until many years after the C++0x standard is finalized.

```
public:
  SomeClass(): isIntialized_(false), member1_(1), member2_(5.0) {}
  void set_A(const RCP<A> &a) { a_ = a; isIntialized_=false; }
  void set_B(const RCP<B> &b) { b_ = b; isIntialized_=false; }
  RCP<A> get_A() { return a_; }
  RCP<B> get_B() { return b_; }
  RCP<C> get_C() { justInTimeInitialize(); return c_; }
  void uninitialize() { a_ = null, b_ = null; c_ = null; isIntialized_=false; }
  void doSomeOperation(...)
    {
      justInTimeInitialize();
      ...
     }
private:
  bool isIntialized_;
  int member1_;
  double member2_;
  RCP<A> a_;
  RCP<B> b_;
  RCP<C> c_;
  void justInTimeInitialize()
    {
      if (isIntialized_) return;
      // Can now call virtual functions on SomeBaseClass (someBaseFunc())!
      if (nonnull(a_) && nonnull(b_))
        c_ = createC(rcp(this, false), a, b, this->getSomeValue());
      ...
      isIntialized_ = true;
    }
};

// Non-member constructors
RCP<SomeClass> someClass()
  { return rcp(new someClass()); }
RCP<SomeClass> someClass(const RCP<A> &a)
  { RCP<someClass> sc(new someClass()); sc->set_A(a); return sc; }
RCP<SomeClass> someClass(const RCP<B> &b)
  { RCP<someClass> sc(new someClass()); sc->set_B(b); return sc; }
RCP<SomeClass> someClass(const RCP<A> &a, const RCP<B> &b)
  { RCP<someClass> sc(new someClass()); sc->set_A(a); sc->set_B(b);  return sc; }
```

SIDE NOTE: Before describing the specific advantages of the refactored class in Listing 112, first note that the issue of the creation of the `C` object and dangling references of `c_` returned from `get_C()` have not been addressed in this design. That issue will be addressed with the "object self-reference" idiom described in Section 5.13.3.

Some of the specific advantages of the usage of the "separate construction and initialization" idiom as applied to the design of `SomeClass` shown in Listing 112 are described below.

a) The default values for `member1_` and `member2_` are defined in only one constructor initialization list. This massively simplifies the maintenance of large complex classes with lots of data members and more than one constructor.

b) The private initialization function `justInTimeInitialize()` can now call a virtual function on the base class `SomeBaseClass::getSomeValue()` to get the value of `someValue` instead of requiring the client to pass it into the constructor.

c) The objects `a` and `b` can be constructed and injected into the `SomeClass` object in different parts of the code by different clients. This breaks a fundamental dependency which couples these objects and the clients together and can massively simplify the structure of complex programs.

d) The "separate construction and initialization" idiom naturally leads to the "just-in-time initialization" idiom where the `justInTimeInitialize()` function is not called until the functions `doSomeOperation(...)` or `get_C()` are called by a client. This allows the objects `a` and/or `b` to be passed into the functions `set_A(...)` and `set_B(...)` in a partially initialized state. These objects do not need to be fully initialized until the `doSomeOperation(...)` or `get_C()` functions are called. This can massively simplify and robustify the design of complex programs by separating code that creates the links between objects from the code that fully initializes the objects. This avoids the constraints of needing to use a factory object to create fully initialized "aggregate" objects described in [14].

e) An object of type `SomeClass` is not any harder for a client to create because the non-member constructor functions allow the object to be constructed in a single function call (see Section 5.8.1) for all the use cases given in the original class constructor design.

The only real disadvantage of the "separate construction and initialization" idiom is some small decrease in performance in using assignment instead of member initialization lists [26, Item 4]. However, this type of low-level performance is almost never an issue in higher-level classes like `SomeClass` shown in Listing 112. Most classes in a complex program are higher-level classes where low-level performance considerations like this are not an issue so the the "separate construction and initialization" idiom is applicable in more cases than not.

The main disadvantage of the "just-in-time initialization" idiom is the need to have a call the function `justInTimeInitialize()` in every operation that requires the object to be fully initialized. This is minor programming inconvenience and a minor performance overhead. The more significant disadvantage is that more unit testing is needed to test the behavior of the user functions for when the object is not ready to be fully initialized. However, good class design makes this fairly easy.

### 5.13.3 The object self-reference idiom

There are occasions where an object needs to provide an `RCP` to itself with the full protection of the debug-mode checking with node-tracing enabled. However, for an object to hold a strong `RCP` to itself would set up a circular reference and the object would never be deleted. The issue of self references was mentioned in the previous section in the context of the "separate construction and initialization" idiom.

The most straightforward example of where the "object self-reference" idiom is needed is when a factory object creates a product that must in turn store a strong owning `RCP` to the factory that created it. This is the exact use case that exists in the Thyra package for `VectorBase` and `VectorSpaceBase` objects [3]. In this case, `VectorSpaceBase` acts as the factory and `VectorBase` acts as the product. Also, every `VectorBase` object has a function `space()` that returns an `RCP` to the `VectorSpaceBase` object that created it to be used to create other `VectorBase` objects.

A simplified version of the implementation of the `VectorSpaceBase` standard subclass `DefaultSpmdVectorSpace` using the "object self-reference" idiom is shown in Listing 113.

**Listing 113** *: Example of the "object self-reference" idiom where a factory must give a strong owning `RCP` self reference to its products.*

```
class DefaultSpmdVectorSpace : public VectorSpaceBase {
  RCP<DefaultSpmdVectorSpace> weakSelfPtr_;
  Ordinal localDim_;
  DefaultSpmdVectorSpace() : localDim_(0) {}
public:
  static RCP<DefaultSpmdVectorSpace> create()
    {
      RCP<DefaultSpmdVectorSpace> vs(new DefaultSpmdVectorSpace);
      vs.weakSelfPtr_ = vs.create_weak();
      return vs;
    }
  void initialize(const Ordinal localDim)
    {  localDim_ = localDim; }
  RCP<VectorBase> createMember()
    { return defaultSpmdVector(weakSelfPtr_.create_strong()); }
};

// Nonmeber constructor
RCP<DefaultSpmdVectorSpace> defaultSpmdVectorSpace(const Ordinal localDim)
{
  RCP<DefaultSpmdVectorSpace> vs = DefaultSpmdVectorSpace::create();
  vs->initialize(localDim);
  return vs;
}
```

The way the "object self-reference" idiom works is that a static function `create()` allocates a default-initialized `DefaultSpmdVectorSpace` object and stores it in a strong owning `RCP` object. It then creates a weak `RCP` object that it sets as the self reference on the newly created `DefaultSpmdVectorSpace` object. The default constructor is made private so the only way for a client to create an `DefaultSpmdVectorSpace` object is to use use the static `create()` function (or call it indirectly through the non-member constructor function `defaultSpmdVectorSpace()`). Because this self reference is a weak tracing `RCP`, it can detect dangling references or can be used to create a strong `RCP` when needed while at the same time not creating a circular reference that would result in a memory leak.

The member function `createMember()` shown in Listing 113 creates a strong owning `RCP` to itself which is given to the newly created `DefaultSpmdVector` object in the statement `defaultSpmdVector(weakSelfPtr_.create_strong())`. This allows the resulting product `DefaultSpmdVector` object to outlive the client's `RCP` references to the `DefaultSpmdVectorSpace` factory object. A simple example of this is shown in Listing 114.

**Listing 114** *: Example of client code that creates a factory, uses it to create a product and lets the factory go away where the factory is remembered in the product*

```
RCP<VectorBase> createMyVector(const Ordinal localDim)
```

```
{
   RCP<DefaultSpmdVectorSpace> vs = defaultSpmdVectorSpace(localDim);
   return vs->createMember();
   // NOTE: The DefaultSpmVectorSpace object is embedded in the returned
   // DefaultSpmdVector object and will not be deleted
}
```

Code like shown in Listing 114 may seem contrived but there have been several use cases for Thyra over the years that required code just like this to work or it would have resulted in a much more complex design of the client's code to work around this issue.

There are also other less obvious examples where the "object self-reference" idiom is useful. For one such case, consider Listing 115 which shows a simplified version of the class shown in Listing 111 that has to pass a self reference to an object it creates and holds internally.

**Listing 115** *: Example of an class with `RCP`-to-self problems (similar to the class in Section 5.13.2)*

```
class SomeClass : public SomeBaseClass {
  RCP<C> c_;
  void finalInitialization() { ...}
public:
  SomeClass() {}
    { c_ = createC(rcp(this, false)); finalInitialization(); }
  RCP<C> get_C() { return c_; }
  ...
};
```

The problem with the code in Listing 115 is that it gives up an `RCP<C>` object to its internal `C` object that is constructed internally but a proper node tracing relationship has not been established between the `C` object and the `SomeClass` object. (Even if `SomeClass` does not intentionally give up its `RCP<C>` object, it is still very easy to do it by accident so this scenario still applies.) To see the problem with this, consider the client code in Listing 116.

**Listing 116** *: Client code that results in undefined behavior (e.g. segfault)*

```
RCP<SomeClass> sc(new SomeClass);
RCP<C> c = sc->get_C();
sc = null; // The SomeClass object is destroyed which invalidates 'c'!
c->someFunc(); // Undefined behavior of call to deleted SomeClass!
```

The problem with the code in Listing 116 is that when the `SomeClass` object `sc` is destroyed, there is no way for the reference-counting machinery in to catch the dangling reference. This code yields undefined behavior and will segfault if one is lucky but like any memory usage error, if one is unlucky the code will appear to work correctly but will be a ticking time-bomb that will go off eventually. The reason for this behavior is that the statement `c_ = createC(rcp(this, false))` in the constructor `SomeClass()` creates a non-owing `RCPNode` object before the owning `RCPNode` object is created by the client code `RCP<SomeClass> sc(new SomeClass)`. This violates Commandment 5 in Appendix B. The node-tracing

reference-counting machinery would have to be more complex and much more expensive to catch dangling reference errors where the strong owning `RCPNode` object was not the first `RCPNode` object created.

The solution to this problem is to use a variation of the "object self-reference" idiom. The updated design that accomplishes this is shown in Listing 117.

**Listing 117** *: Example of the "object self-reference" idiom for detecting dangling references to internally held objects*

```
class SomeClass : public SomeBaseClass {
  RCP<SomeClass> weakSelfPtr_;
  RCP<C> c_;
  SomeClass() {}
  void justInTimeInitialize() { c_ = createC(weakSelfPtr_); }
public:
  static RCP<SomeClass> create()
    {
      RCP<SomeClass> sc(new SomeClass);
      sc.weakSelfPtr_ = sc.create_weak();
      return sc;
    }
  RCP<C> get_C() { justInTimeInitialize(); return c_; }
  ...
};

// Nonmeber constructor
RCP<SomeClass> someClass() { return SomeClass::create(); }
```

The advantage of the design in Listing 117 is that now client code like shown in Listing 118 below will result in a dangling reference exception in a node-tracing debug-mode build.

**Listing 118** *: Client code that results a clean dangling-reference exception*

```
RCP<SomeClass> sc = someClass();
RCP<C> c = sc->get_C();
sc = null; // The SomeClass object is destroyed which invalidates 'c'!
c->someFunc(); // Now the dangling reference is detected and throws!
```

Note that the implementation of `SomeClass` in Listing 117 means that the nature of the relationship between the `C` object returned from `SomeClass::get_C()`, the parent `SomeClass` object, and the client code represents a semi-persisting association as defined in Section 4.2. In this case, the usage of the `C` object is only valid while the parent `SomeClass` object still exists. However, if the client mistakenly tries to use a dangling `C` object after its parent `SomeClass` object is destroyed, then a clean runtime dangling-reference exception is thrown as described in Section 5.11.3.

Alternatively, if one wants the `SomeClass::get_C()` function to create a true persisting association where the `C` object can outlive all of the client references to the parent `SomeClass` object, then the implementation of `SomeClass::get_C()` can be modified to what is shown in Listing 119.

**Listing 119** *: Implementation of the "object self-reference" idiom using a true persisting association*

```
RCP<C> SomeClass::get_C()
{
  justInTimeInitialize();
  return rcpInvertedObjOwnership(c, weakSelfPtr_.create_strong());
}
```

Creating a strong `RCP` is required in this case as well as the inversion of the object ownership (which is an instance of the "inverted object ownership" idiom which is described in Section 5.13.1).

Given the implementation in Listing 119, client code like shown in Listing 118 will allow the `C` object to be used after the client's `RCP<SomeClass>` object is made `null` without thrown a dangling reference exception. Choosing between a semi-persisting or a persisting association is a design decision for the creator of `SomeClass`.

### 5.13.4 The generalized view design pattern

One of the most useful and powerful idioms / design-patterns related to the use of the Teuchos memory management classes is the "generalized view" design pattern[18]. In this context, a "view" is some object that is created off of a parent object and provides some type of access to some part of the parent. Views can be const or non-const and can be persisting or semi-persisting (see Section 4.2 for the definition of persisting and semi-persisting associations). Views can also be direct views or "generalized views" (i.e. potentially detached non-direct views).

A direct view is one which directly points into the the internal data structures of the parent so that a change of the view instantaneously changes the parent and changes to the parent instantaneously changes the view(s). An example of a direct view is an iterator into an container such as is returned from `std::vector::begin()` or `std::list::begin()`. Other examples of direct views include `ArrayView` views of `Array` and `ArrayRCP` objects. Direct views can be non-const or const as is demonstrated with iterators and `ArrayViews`. Direct views are a pleasure to work with but they also fundamentally constrain the implementation of the parent objects that they are providing the views into. In the case of contiguous array containers like `std::vector` and `ArrayRCP`, constraining the implementation to store a pointer to a contiguous array of data internally is not a problem, that is an important and proper design constraint for these classes. However, for more general classes, the rigid constraints imposed by direct views are unacceptable and break the abstraction in many cases. For example, if an abstract matrix object provides direct views of the rows of a matrix then the matrix must necessarily be stored in a row-major data-structure, precluding other possibilities. Once a direct row-based view is supported by such a matrix class, it becomes impossible to change the internal representation of the matrix to anything other than a row-oriented implementation and still maintain high performance and a reasonable implementation.

[18] Here the term 'design pattern' and not 'idiom' is used to describe the "generalized view" design pattern. The reason that the more general term 'design pattern' is being used is that the majority of the pattern is really language independent and the behaviors are more general then what one will find in a typical language-specific idiom. It is only the `RCP` details that would classify this as a C++ idiom. However, this is an important example of the use of `RCP` so it is worthy to be discussed in this document.

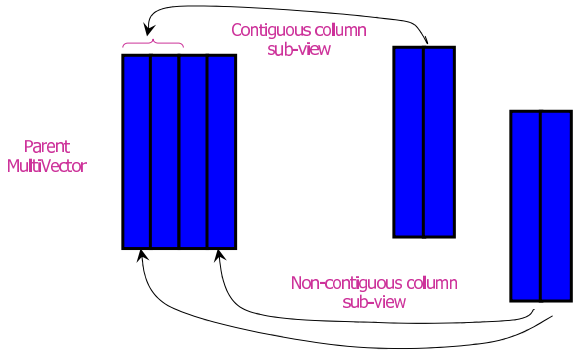



**Figure 15.** Depiction of contiguous and non-contiguous multi-vector column views.

### Basic overview of the "generalized view" design pattern

Because direct views can overly constrain the implementation freedom of the parent, in order to allow for the fullest freedom to pick the internal implementations of the parent and the view separately, we must instead consider using potentially detached non-direct views defined here as "generalized views". A "generalized view" is a view of a parent object that is not guaranteed to be a direct view such that changes to the view may not be instantaneously propagated to the parent and vice versa. One example of a case where a "generalized view" is needed is when creating a view of non-contiguous columns of a dense matrix where, for the sake of efficiency, one must create a temporary contiguous copy as a new dense matrix. This type of generalized view is used in the implementation of Thyra MultiVector non-contiguous column views [3] which is depicted in Figure 15. In the Thyra MultiVector implementation, when the client requests a view of a set of non-contiguous columns, the implementation will create a temporary contiguous copy which results in improved performance of many types of operations[19]. The Thyra example of non-contiguous column views is a good example because it is a simple case to describe yet has all the features needed to demonstrate the workings of the generalized view design pattern.

Before describing the Thyra MultiVector example in more detail, first, a generic description of the "generalized view" design pattern is presented. The most general description of the "generalized view" design pattern is shown in Figure 16 (UML class diagram) and Figure 17 (UML state diagram). Figure 16 shows two generic classes; a `Parent` and a `View`. In this case, only one type of the view is shown but in reality there can be several different types of views into a single parent object (as there are in the Thyra MultiVector case). A view is created using either the `createNonconstView(...)` or the `createView(...)` functions. In either case, the returned view is wrapped in an `RCP` object. There are two purposes for wrapping the view in an `RCP` object. First, a new `View` object may need to be dynamically allocated to satisfy the view request (and therefore needs `RCP` to take care to control the lifetime of the dynamically allocated view object). Second, the action needed to re-sync the parent up with the view can be set as an embedded object on the returned `RCP` (see Section 5.9.4). In fact, the `Parent` object is not guaranteed to be updated after changes to a non-const view are made until the view is destroyed/released. To demonstrate, consider the client code in Listing 120.

[19] Using contiguous columns of a Fortran-style column-major dense matrix is required in order to take advantage of high performance Basic Linear Algebra Subroutines (BLAS) software [11].

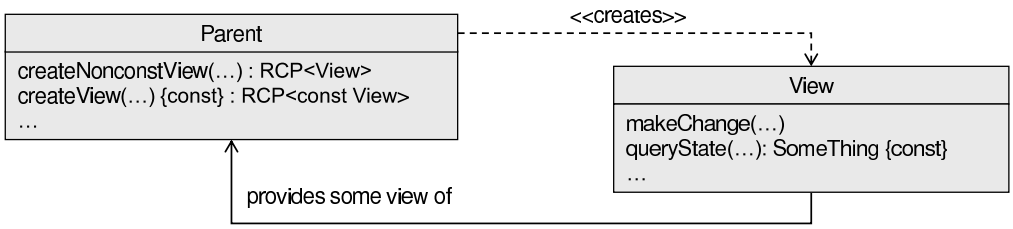



**Figure 16.** Parent and child classes for "generalized view" design pattern.

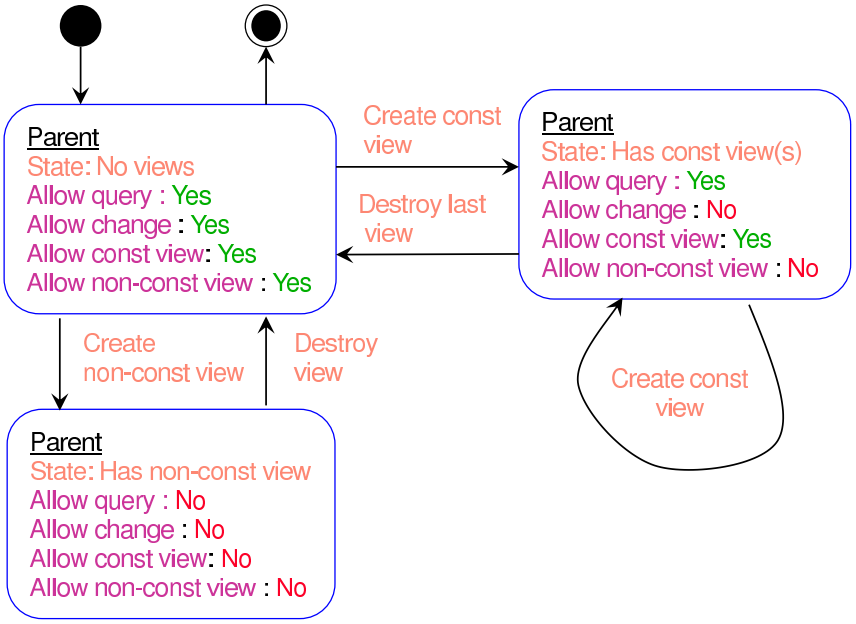



**Figure 17.** State behavior for parent object in "generalized view" design pattern.

**Listing 120** *: Example use of a generalized view*

```
void changeParent(const Ptr<Parent> &parent)
{
  // Create a non-const view
  const RCP<View> view = parent->createNonconstView(...);

  // Change the parent through the view
  view->makeChange(...);
  view->makeChange(...);
  ...

  // Destroy the view which resycs the view with the parent
  view = null;

  // Now the parent has been updated for changes through the view!

}
```

The code in Listing 120 demonstrates that the parent is only guaranteed to be updated after the non-const generalized view has been destroyed. The state of the parent is undefined while a non-const view is active. One of three possibilities exist for the state of the parent while a non-const view is active. First, if the view just happens to be a direct view, then changes to the view are instantaneously reflected in the parent. Second, if the view is a completely separate chunk of data, changes to the view do not affect the state of the parent at all until the view is destroyed and the data in the view is copied into the parent's internal data structures in the appropriate way. The third option is somewhere in between the first two; part of the view's data may directly point back into the parent's internal data structures and the rest of the view's data may be separate from the parent. In this case, the state of the parent object may actually violate its internal invariants and the parent object would not even be usable while the non-const view is active.

In order to allow complete freedom in how a generalized view is implemented and to allow the implementation to change at will, a relatively strict usage protocol must be defined as shown in the parent's state diagram in Figure 17. With respect to generalized views, the `Parent` has one of three states; 'no views', 'has const views(s)', and 'has non-const view'. The default state of the parent is 'no views'. In this state, either a const or a non-const view can be created and all types of query and modifying functioins can then be called. When a const view is created, the parent enters the state 'has const view(s)'. In this state, the parent object is still allowed to be queried and other const-views can be created. However, while there are active const views, a non-const view cannot be created. The reason that non-const views cannot be allowed while const views are active is that creating a non-const view and then changing it while const views are active would put the const views into an undefined state. If the const and non-const views happened to be implemented as direct views, then changes to the non-const view would not only change the parent but it would also change the direct const views. However, if the views are implemented as detached copies of data, then changes to the non-const view would not be expected to be propagated to the existing const views[20]. Since this type of ambiguity would destroy the abstraction, the generalized view design pattern

[20] Actually, one could implement an OBSERVER [23] type of implementation where changes to the views would automatically be written back and forth to keep the parent and the views in sync but this would lead to complex and fragile implementations and could significantly degrade performance if frequent small changes to data resulted in lots of syncs. Therefore, to allow for a simple implementation and the highest performance, the generalized view design pattern discourages this type of more complex less efficient OBSERVER-type of implementation.

simply states that creating non-const views on the parent while const views are active is just not allowed. Likewise, any operations on the parent that might change the state in a way that would affect the views must also be disallowed. In summary, when const views are active, the non-const interface of the parent must be locked down. When the last const view is destroyed, the parent goes back to the 'no views' state.

When a non-const view is created the parent goes into the 'has non-const view' state. In this state, no other non-const or const views can be created and the entire const and non-const interface of the parent must be locked down. In essence, when a non-const view is active, the parent object has to be completely left along. The reason for this should be obvious. When a non-const view is represented as a separate copy of data, then the state of the parent is undefined until the non-const view is destroyed and the data is written back. In this case, a query of the parent would not show the changes made in the active non-const view. Again, this type of ambiguity would destroy the abstraction and therefore the parent object must be totally locked down while a non-const view is active. Likewise, only one non-const view can be allowed at any one time due to similar arguments. When the non-const view is destroyed, any changes in the data are written back to the parent's internal data structures and the parent goes back to the state 'no views'.

In order to allow for the highest performance, the simplest implementations of the parent and the view classes in all cases, and to help catch errors in client code, therefore by default the generalized view design pattern states that views should be semi-persisting; that is, the views are only valid while the client's `RCP` to the parent is still active and the view is not expected to live on past the lifetime of the parent. Therefore, any access to generalized views that remain after the last strong `RCP<Parent>` object is released should result in dangling reference exceptions in a debug-mode build as shown in Listing 121.

**Listing 121** *: Example of a dangling reference generalized view*

```
// Create and initialize parent
RCP<Parent> parent = createParent(...);
...

// Create a view
RCP<View> view = parent->createNonconstView(...);

// Destroy the parent (invalidating the existing view)
parent = null;

// Try to access the now dangling view
view->makeChange(...); // Throws dangling reference exception!
```

Even if a generalized view can be made persisting (which it is in the MultiVector example given later), the implementation should still prefer to implement non-const views as semi-persisting views. The main purpose of a non-const view is to change the parent object so if the parent is released before the non-const view is written to and written back, then that is most likely a programming error in the client's code. For example, the code in Listing 121 is most likely an error in program logic because if the parent has been deleted then there is no use in modifying the view. By implementing generalized views and semi-persisting views the objects help to better catch these types of errors in client code.

However, if it makes sense in the particular setting, an implementation of the generalized view design pattern can choose to implement the views as full persisting views that will persist even after all the external parent references are removed (thereby avoiding dangling-reference exceptions). Note that if the

parent class implements the "object self-reference" idiom described in Section 5.13.3 then a strong `RCP` to the parent object can always be attached to the `RCP` of the view, thereby providing for persisting generalized views no mater what internal data structures are used.

Another aspect of the generalized view design pattern is that it is important to distinguish between non-const views and const views. In the case of const views, if a separate copy of the data must be created to support the const view, the data does not have to be written back to the parent when the const view is destroyed. This makes const views fundamentally more efficient than non-const views (which must write back their data when they are destroyed). Because const views are potentially more efficient, the unadorned name `createView(...)` is given to create const views while the longer name `createNonconstView(...)` is used to create non-const views. The idea is that a developer is more likely to call the shorter `createView(...)` function creating a more efficient and safe const view. If a const view is all the client code requires, then all is good. However, if the client code really needs to change the parent object through the view, then the code will not compile and the developer will need to change the client code to create a non-const view through `createNonconstView(...)`.

Because the parent object gets locked down (or is in an undefined state when debug usage checking is not enabled) when generalized views are active, it is important that client code only create views at the last possible moment and then release them at the earliest possible moment. The best way to do this, when possible, is to create and release the view in the same statement where the view will be used as demonstrated in Listing 122.

**Listing 122** *: Example of minimizing the lifetime of a generalized view*

```
void changeParentThroughView(const Ptr<Parent> &parent)
{
  changeTheView(*parent->createNonconstView());
  queryTheView(*parent->createView());
  ...
}
```

The client code in Listing 122 works just fine because the temporary `RCP<[const] View>` objects managed by the compiler are guaranteed to exist until the full statement they are created in ends. This is one case where using an `RCP` to manage the memory is very convenient.

The last issue to discuss related to the generalized view design pattern is that, depending on the nature of the parent and the view classes, it may be reasonable to have the parent object partitioned into different logical pieces and then apply the behaviors shown in Figure 17 to each of these logical pieces separately. For example, one can treat each row or each column in a matrix object as a separate logical piece such that one can allow separate views of each of the rows or columns independent of each other. This is the case with the Thyra MultiVector example (mentioned earlier and to be described in more detail below). However, any bulk query operations on the parent object (like taking an induced matrix norm) must be locked out while non-const views are active. Likewise, any bulk modifying operation (like assigning all the matrix entries to zero) must be locked out when any views are active.

**Example implementation of generalized views for MultiVector non-contiguous column views**

Now that a basic overview of the generalized view design pattern has been given, the Thyra MultiVector non-contiguous column view example mentioned earlier and depicted in Figure 15 is described in more detail. This is a good example to highlight the features of the generalized view design pattern and the usage of the `RCP` class to manage detached view semantics. A simplified class declaration for the Thyra MultiVector subclass showing the relevant class members is given in Listing 123.

**Listing 123** *: Class declaration for Thyra MultiVector implementation of multi-vector views as "generalized views"*

```
template<class Scalar>
class DefaultSpmdMultiVector : virtual public SpmdMultiVectorBase<Scalar> {
public:

  ...

  DefaultSpmdMultiVector(
    const RCP<const SpmdVectorSpaceBase<Scalar> > &spmdRangeSpace,
    const RCP<const ScalarProdVectorSpaceBase<Scalar> > &domainSpace,
    const ArrayRCP<Scalar> &localValues,
    const Ordinal leadingDim = -1
    );

  ...

protected:

  ...

  RCP<const MultiVectorBase<Scalar> >
  nonContigSubViewImpl(const ArrayView<const int> &cols) const;

  RCP<MultiVectorBase<Scalar> >
  nonconstNonContigSubViewImpl(const ArrayView<const int> &cols);

  ...

private:

  RCP<const SpmdVectorSpaceBase<Scalar> > spmdRangeSpace_;
  RCP<const ScalarProdVectorSpaceBase<Scalar> > domainSpace_;
  ArrayRCP<Scalar> localValues_;
  Ordinal leadingDim_;

  ArrayRCP<Scalar> createContiguousCopy(const ArrayView<const int> &cols) const;

};

// Non-member constructor
template<class Scalar>
RCP<DefaultSpmdMultiVector<Scalar> >
```

```
defaultSpmdMultiVector(
  const RCP<const SpmdVectorSpaceBase<Scalar> > &spmdRangeSpace,
  const RCP<const ScalarProdVectorSpaceBase<Scalar> > &domainSpace,
  const ArrayRCP<Scalar> &localValues,
  const Ordinal leadingDim = -1
  )
{
  return Teuchos::rcp(
     new DefaultSpmdMultiVector<Scalar>(
      spmdRangeSpace, domainSpace, localValues, leadingDim ) );
}
```

The internal private data-structure for a multi-vector is very simple as shown in Listing 123. A standard column-major Fortran-style dense matrix format is used where all of the data in the local processes is stored in a single contiguous `ArrayRCP<Scalar>` object. The number of rows in the local process is given by `spmdRangeSpace_->localSubDim()` and `leadingDim_` is the stride between columns.

The generalized views returned by the functions `nonContigSubViewImpl(...)` and `nonconstNonContigSubViewImpl(...)` are of the type `MultiVectorBase` which is the upper-most base class for `DefaultSpmdMultiVector`. (The concrete types of the views are actually `DefaultSpmdMultiVector`.) Therefore, this is an instance where the class types of the parent and view are actually the same (an interesting example of CLOSURE OF OPERATIONS principle [14, Chapter 10])!

The implementations of the functions `nonContigSubViewImpl(...)` and `nonconstNonContigSubViewImpl(...)` are given in Listing 124.

**Listing 124** *: Implementation of `DefaultSpmdMultiVector` functions `nonContigSubViewImpl(...)` and `nonconstNonContigSubViewImpl(...)`*

```
template<class Scalar>
RCP<const MultiVectorBase<Scalar> >
DefaultSpmdMultiVector<Scalar>::nonContigSubViewImpl(
  const ArrayView<const int> &cols ) const
{
  THYRA_DEBUG_ASSERT_MV_COLS("nonContigSubViewImpl(cols)", cols);
  const int numCols = cols.size();
  const ArrayRCP<Scalar> localValuesView = createContiguousCopy(cols);
  return defaultSpmdMultiVector<Scalar>(
    spmdRangeSpace_,
    createSmallScalarProdVectorSpaceBase<Scalar>(*spmdRangeSpace_, numCols),
    localValuesView );
}


template<class Scalar>
RCP<MultiVectorBase<Scalar> >
DefaultSpmdMultiVector<Scalar>::nonconstNonContigSubViewImpl(
  const ArrayView<const int> &cols )
{
  THYRA_DEBUG_ASSERT_MV_COLS("nonContigSubViewImpl(cols)", cols);
  const int numCols = cols.size();
```

```
  const ArrayRCP<Scalar> localValuesView = createContiguousCopy(cols);
  const Ordinal localSubDim = spmdRangeSpace_->localSubDim();
  RCP<CopyBackSpmdMultiVectorEntries<Scalar> > copyBackView =
    copyBackSpmdMultiVectorEntries<Scalar>(cols, localValuesView.getConst(),
      localSubDim, localValues_.create_weak(), leadingDim_);
  return Teuchos::rcpWithEmbeddedObjPreDestroy(
    new DefaultSpmdMultiVector<Scalar>(
      spmdRangeSpace_,
      createSmallScalarProdVectorSpaceBase<Scalar>(*spmdRangeSpace_, numCols),
      localValuesView),
    copyBackView );
}
```

The implementation of the sub-view functions in Listing 124 is fairly simple. First, the private helper function `createContiguousCopy(...)` creates an `ArrayRCP<Scalar>` object for a contiguous copy of the non-contiguous columns being requested. This contiguous copy of data is then given over to create a new `DefaultSpmdMultiVector` object which represents the view. The implementation of the function `createContiguousCopy(...)` is simple enough and is given in Listing 125.

**Listing 125** *: Implementation of `DefaultSpmdMultiVector` function `createContiguousCopy(...)`*

```
template<class Scalar>
ArrayRCP<Scalar>
DefaultSpmdMultiVector<Scalar>::createContiguousCopy(
  const ArrayView<const int> &cols ) const
{
  typedef typename ArrayRCP<Scalar>::const_iterator const_itr_t;
  typedef typename ArrayRCP<Scalar>::iterator itr_t;
  const int numCols = cols.size();
  const Ordinal localSubDim = spmdRangeSpace_->localSubDim();
  ArrayRCP<Scalar> localValuesView = Teuchos::arcp<Scalar>(numCols*localSubDim);
  // Copy to contiguous storage column by column
  const const_itr_t lv = localValues_.begin();
  const itr_t lvv = localValuesView.begin();
  for (int k = 0; k < numCols; ++k) {
    const int col_k = cols[k];
    const const_itr_t lv_k = lv + leadingDim_*col_k;
    const itr_t lvv_k = lvv + localSubDim*k;
    std::copy(lv_k, lv_k+localSubDim, lvv_k);
  }
  return localValuesView;
}
```

Note how iterators are used to perform the raw data copy in Listing 125. This results in very well checked code in a debug-mode build (see Section 5.11) but very high performance code in a non-debug optimized build (see Section 5.12).

Note that the key difference between the implementation of the functions `nonContigSubViewImpl(...)` and `nonconstNonContigSubViewImpl(...)` in Listing 124 is that

`nonconstNonContigSubViewImpl(...)` creates an `RCP` to an object of type `CopyBackSpmdMultiVectorEntries` and attaches it to the created `RCP<DefaultSpmdMultiVector<Scalar> >` object as an embedded object (see Section 5.9.4). The destructor for `CopyBackSpmdMultiVectorEntries` performs the copy-back of the non-const view after the last `RCP` to the view is destroyed. The implementation of the class `CopyBackSpmdMultiVectorEntries` is given in Listing 126.

**Listing 126** *: Implementation of the class `CopyBackSpmdMultiVectorEntries`*

```
template<class Scalar>
class CopyBackSpmdMultiVectorEntries {
public:
  CopyBackSpmdMultiVectorEntries(
    const ArrayView<const int> &cols,
    const ArrayRCP<const Scalar> &localValuesView, const Ordinal localSubDim,
    const ArrayRCP<Scalar> &localValues, const Ordinal leadingDim
    )
    : cols_(cols), localValuesView_(localValuesView), localSubDim_(localSubDim),
      localValues_(localValues), leadingDim_(leadingDim)
    {}
  ~CopyBackSpmdMultiVectorEntries()
    {
      typedef typename ArrayRCP<const Scalar>::const_iterator const_itr_t;
      typedef typename ArrayRCP<Scalar>::iterator itr_t;
      // Copy from contiguous storage column by column
      if (localValues_.strong_count()) {
        const int numCols = cols_.size();
        const const_itr_t lvv = localValuesView_.begin();
        const itr_t lv = localValues_.begin();
        for (int k = 0; k < numCols; ++k) {
          const int col_k = cols_[k];
          const const_itr_t lvv_k = lvv + localSubDim_*k;
          const itr_t lv_k = lv + leadingDim_*col_k;
          std::copy( lvv_k, lvv_k + localSubDim_, lv_k );
        }
      }
    }
private:
  Array<int> cols_;
  ArrayRCP<const Scalar> localValuesView_;
  Ordinal localSubDim_;
  ArrayRCP<Scalar> localValues_;
  Ordinal leadingDim_;
};


// Non-member constructor
template<class Scalar>
RCP<CopyBackSpmdMultiVectorEntries<Scalar> >
copyBackSpmdMultiVectorEntries(
  const ArrayView<const int> &cols,
  const ArrayRCP<const Scalar> &localValuesView, const Ordinal localSubDim,
```

```
  const ArrayRCP<Scalar> &localValues, const Ordinal leadingDim
  )
{
  return Teuchos::rcp(
    new CopyBackSpmdMultiVectorEntries<Scalar>(
      cols, localValuesView, localSubDim, localValues, leadingDim));
}
```

The implementation of `CopyBackSpmdMultiVectorEntries` in Listing 126 is straightforward. When the destructor is called, it copies the data in the non-const view back to the native storage of the parent `DefaultSpmdMultiVector` object.

The only twist in the implementation of `nonconstNonContigSubViewImpl(...)` and `CopyBackSpmdMultiVectorEntries` is that a weak `ArrayRCP` is used for the parent's `localValues_` data in the `CopyBackSpmdMultiVectorEntries` object. It is created in the function `nonconstNonContigSubViewImpl(...)` with `localValues_.create_weak()`. If the parent goes away before the view, then the weak pointer `localValues_` in the destructor for `CopyBackSpmdMultiVectorEntries` will have a strong count of 0, thereby resulting in the skipping of the copy-back of data. This this a performance optimization since there is no point in copying back the data if the parent object is gone. This design allows both const and non-const multi-vector views to be persisting (past the lifetime of the parent) and still have the highest performance.

There are several things that are interesting about this example. First, by using the embedded object feature of `RCP`, the code is able to implement the copy-back-to-parent functionality without having to write a new `MultiVector` subclass just for the view. The `DefaultSpmdMultiVector` objects that are returned as views have no idea they are being used as views into other `DefaultSpmdMultiVector` objects. Without the embedded object feature described in Section 5.9.4, a different `MultiVector` subclass would have to be created with a destructor that would copy back the data. Second, the constraints imposed by the generalized view design pattern shown in Figure 17 ensure that no problems will arise due to the fact that the views are stored in detached copies of the data. If changes to the parent `DefaultSpmdMultiVector` were allowed while a non-const `DefaultSpmdMultiVector` view was active, then all of the changes in the parent would be overwritten when the `DefaultSpmdMultiVector` view was destroyed. Third, this example demonstrates why const views are fundamentally more efficient than non-const views. In the case of a const view, the temporary contiguous copy of data is just released and does not need to be copied back to the parent. This saves the work imposed by the destructor on the `CopyBackSpmdMultiVectorEntries` object.

#### Summary of the generalized view design pattern

In summary, the main properties and features of the generalized view design pattern are:

- Generalized views allow for complete abstraction, encapsulation, and the highest performance in all cases. This is simply not possible to achieve with direct views.
- Non-const generalized views are only guaranteed to update the state of the parent after the view object has been released.
- A single parent object can provide more than one type of generalized view and views can be applied separately to different logically distinct parts of the parent object (e.g. views to the rows or columns

of an abstract matrix object can be created and handled separately).

- It is important to differentiate between non-const views and const views. While detached non-const views must be copied back to parent when the view is released, const views do not (therefore improving the performance of const views).
- It is critical that `RCP` objects be used to wrap the created view objects in order to allow the views to be dynamically allocated and to allow for specialized copy-back behavior when non-const views are destroyed.
- The flexibility and performance gains allowed by the generalized view design pattern come at the expense of more restricted usage patterns of the parent and view objects.
  - Only one non-const view can be active for any logically district part of the parent object at any one time and the parent object (or at least any functionality that relates to the viewed part) must be locked down while a non-const view is active.
  - Multiple const views can be active for any logically district part of the parent object at any one time but the non-const interface of the parent object (at least any non-const functionality that relates to that viewed part) must be locked down while any const views are active.

## 5.14 Comparison with other class libraries and the standard C++ library

Comparisons between the Teuchos memory management classes and other classes in Boost and the standard C++ library have been made throughout this document. Here, these comparisons are summarized and extended. Comparisons with Boost classes are for version 1.40.

The Teuchos class `RCP` is almost identical in most respects to the `boost::shared_ptr` class and therefore also the `std::tr1::shared_ptr` in C++03 and `std::shared_ptr` class in C++0x. The first version of the class `RCP` was developed back in 1998 under the name `MemMngPack::ref_count_ptr` as part of the development of the rSQP++ package [32] (now called MOOCHO [4]). At that time, there was no general purpose high-quality reference-counted smart pointer class available and many compilers at the time (e.g. MSVC++ 6.0) could not even support template member functions needed for implicit smart-pointer conversions. After 1998, the first `boost::shared_ptr` class appeared (which did not allow a customized deallocation policy and was therefore not very flexible). Over the years, the two classes independently evolved in very similar ways. The current version of `boost::shared_ptr` is a high-quality flexible reference-counted smart pointer class. Because it now supports custom template deallocator policy objects (which are called "deleters" in `boost::shared_ptr`) it allows for great flexibility in how it is used.

The key advantages of the `RCP` class over the current `boost::shared_ptr` class are greater functionality, greater flexibility, and better debug-mode runtime checking. The few of the specific key advantages of the `RCP` class that cannot be replicated with the `boost::shared_ptr` class without changing its design include:

- The `RCP` class has built-in support for debug-mode runtime tracing of reference-counting nodes (Section 5.9.1) which is used to implement a whole host runtime checking including the detection and reporting of a) circular references (Section 5.11.2) and b) multiple owning reference-counted objects (Section 5.11.4).
- The `RCP` class allows the association and retrieval of extra data attached to an already-created reference-counting node object (Section 5.9.5).

- The `RCP` class allows a client to call `release()` to remove deletion ownership from an already-created `RCP` object (the rare need for this is described in Section 5.9.5).
- The `RCP` class has built-in support for both strong and weak reference-counted pointer handles right in the same class (see Section 5.9.2). The `boost::shared_ptr` class uses a separate `boost::weak_ptr` class to represent weak references which is less flexible. The `RCP` approach allows the debug-mode runtime detection and reporting of dangling non-owning references while `boost::shared_ptr` class cannot when using a null deleter (Section 5.11.3).

The key advantages of the current `boost::shared_ptr` class over the current `RCP` class are that it has lower storage and runtime overhead (Section 5.12.1).

Because both the Teuchos `RCP` and `boost::shared_ptr` classes support customized deallocation policy objects, one can embed an `RCP` object in a `boost::shared_ptr` object and vice versa. This is already supported in Teuchos using the overloaded non-member template helper functions `Teuchos::rcp(const boost::shared_ptr<T> &p)` and `Teuchos::shared_pointer(const RCP<T> &rcp)` (see Table 8). This allows the developer to mix and match `RCP` and `boost::shared_ptr` objects in the same code and still have correct memory management. However, since `RCP` has better debug-mode runtime checking and is more flexible it should be preferred to `boost::shared_ptr` in most high-level code. Alternatively, because `boost::shared_ptr` has slightly lower overhead and is present `std::tr1` it also has valid uses. Additionally, of course, one may need to convert back and forth between `RCP` and `boost::shared_ptr` objects to glue together different pieces of separately developed code that use different smart pointer classes. The class `boost::scoped_ptr` is identical to `std::auto_ptr` but does not allow copying or assignment and is therefore safer to use in more limited scopes.

Another smart pointer class for single objects is `std::auto_ptr`. This class does not support sharing and has only the minimal functionality needed to support the Resource Allocation Is Initialization (RAII) idiom [31, Item 13]. Given that reference-counting overhead is low compared to raw allocations and deallocations (see Section 5.12.1) there is little reason to ever use `std::auto_ptr` instead of `RCP` (or `boost::shared_ptr` for that matter) except for perhaps the handling of RAII for small objects.

The Teuchos class `Array` is of course equivalent to `std::vector` by design and uses an `std::vector` internally. The main advantages of using `Array` instead of directly using `std::vector` are a) `Array` has better debug-mode runtime checking and produces better error messages, b) conversion to the other Teuchos array types `ArrayView` and `ArrayRCP` includes full runtime debug-mode detection and reporting of dangling references (which is not possible with with `std::vector`), and c) is more consistent with the usage of the other Teuchos array types. A major difference between `Array` and `std::vector` is that `Array` uses an unsigned integer for its `size_type` (see Appendix C for the justification).

The Teuchos class `ArrayRCP` really has no equivalent class in Boost or the C++0x standard libraries. There is a Boost class called `boost::shared_array` which uses the `boost::shared_ptr` reference-counting machinery and has an overloaded `operator[](size_type)` function but his class does not support iterators (which are critical for safety and performance) and does not support persisting sub-views (see Section 5.5.5).

The Teuchos compile-time sized array class `Tuple` is more or less equivalent to the class `boost::array`. Both contain an iterator interface and other STL compliant functions. The key advantage of `Tuple` is that conversions to the other Teuchos array types `ArrayView` and `ArrayRCP` support full runtime debug-mode detection and reporting of dangling references.

Finally, the Teuchos classes `Ptr` and `ArrayView` have no equivalent in Boost or C++0x. As described throughout this paper, these classes are key to creating C++ code that is maximally self documenting (by distinguishing between persisting and non-persisting associations), maximally safe in terms of debug-mode runtime checking, and while at the same time allowing for the highest performance in non-debug optimized builds. One cannot plug the remaining holes in safety and performance without the `Ptr` and `ArrayView` classes.

What makes the Teuchos memory management classes unique across all other class libraries is that they form a complete coordinated system of types to encapsulate all raw C++ pointers in high-level code while at the same time providing 100% secure debug-mode checking. This is only possible because these classes are developed as a system of types and the level of debug-mode runtime checking that exists is only possible because these types have access to each others private implementation (in some appropriate way). In general, one cannot mix and match Boost, standard C++, and Teuchos classes together at the top level and get safe C++ programs with the full extent of debug-mode runtime checking that the integrated set of Teuchos classes provide. Many examples of this have been given through this document. One example is that if one creates an `ArrayView` object from a `std::vector` object it cannot detect a dangling reference (see Section 5.11.3). However, there are some specialized cases where Boost and standard C++ types can be used safely with the Teuchos memory management classes and some of these cases have already been discussed above (e.g. `RCP` and `boost::shared_ptr` objects can be embedded in each other and deep copies of array objects are always safe).

## 5.15 Advice on refactoring existing software

The easiest way to incorporate the full use of the Teuchos memory management classes is to develop new code and use them from the very beginning. In this mode of development, the debug-mode runtime checking makes development fast and productive with one never seeing a segfault or other memory usage error that comes from undefined behavior. However, the more typical situation is that a large existing code-base must continue to be developed, current code must be modified, and new code must integrated with existing code. For existing code bases, the code will need to be refactored to use the Teuchos memory management classes, replacing the use of raw pointers and raw calls to `new` and `delete` along the way.

While code refactored to use the Teuchos memory management classes will be of higher quality and more productive to work with during further development, there will necessarily be a transition period where the code will be refactored to replace current uses of raw C++ pointers and less-than-safe (or inflexible) memory management approaches. It is not recommended that all work on new capabilities stop and the existing code base be refactored all at once to switch over to the complete use of the Teuchos memory management classes. Instead, the code should be refactored to use the Teuchos memory management classes in small iterative cycles as needed. The highest priority code to refactor are the heavily used major module and class interfaces. It is these major interfaces where mistakes and memory usage problems are most likely to be made. However, rather than break backward compatibility it is wise to provide the safe versions of the interface functions but leave the existing unsafe raw-pointer versions when possible and have them call the new safe versions (by converting between raw pointers to the memory management types as needed). This avoids duplication which simplifies further maintenance and also provides for smooth upgrades of client code to incrementally switch over from raw pointers to the Teuchos memory management classes. The help facilitate the transition of client code, the deprecated raw pointer interface functions and other code can be marked as deprecated on some compilers which generates warning messages while compiling (e.g. GCC's `__attribute__((__deprecated__))`).

While the most critical code to refactor to use the Teuchos memory management types are major module and class interfaces, the next most important software to refactor is any software that needs to be changed or extended. Other code is lower priority to refactor, especially existing well-encapsulated code that uses raw C++ pointers internally that does not need to be changed to add new features any time soon. It is not until such code needs to be changed that it should be refactored to use the safer memory management types (which will make adding new features much easier and safer).

While the final state of code refactored to use the Teuchos memory management types is excellent (as described throughout this document), great care must be exercised in refactoring the software. In general, before any piece of software is refactored to use the Teuchos memory management types, it should first be covered with high-quality unit tests (see [15] for a great treatment on how to add unit tests to existing code bases to facilitate adding new features). The general process that should be followed to refactor existing software to use the Teuchos memory management types includes the following major steps (consistent with the advice in [15]):

1. Break dependencies to allow unit tests to be written
2. Add unit tests to cover behavior of the code to be refactored
3. Refactor the targeted code incrementally to use the Teuchos memory management classes (all the while running the unit tests constantly including using Valgrind and/or Purify to ensure defects are not being created) by:
   (a) Replacing raw pointers internally with Teuchos memory management types until all raw pointers are gone
   (b) Writing new versions of the interface functions in terms of the safer Teuchos memory management types
   (c) Keeping the existing raw-pointer interface functions which are called by the unit tests but have them call the new functions that take the safer memory management types
   (d) Marking the raw-pointer versions of the functions as deprecated as so to facilitate refactoring of client code (e.g. using GCC's `__attribute__((__deprecated__))`)
4. After a unit of code is totally refactored to use the new Teuchos memory management types, the unit test code should be refactored to call the safe interface functions that don't pass raw pointers, thereby removing all raw C++ pointers from the unit test code itself.
5. Selectively refactor client code that can conveniently call the new safe interface functions of the refactored code. (Caution, only do this if there are at least some decent system-level regression tests in place.) As more and more code is refactored to use the safe Teuchos memory management types, the easier and safer this type of refactoring will become.
6. Write new unit tests (using test-driven development) and add the desired new features in the selected code safely and easily

While the incremental refactoring process described above may be slow and may only refactor small parts of the code in each batch, over time, more and more of the code base will be refactored to remove raw C++ pointers and the code will become more and more safe, easier to work with, and be better self documenting. Whatever happens, one should never attempt to refactor a large volume of code in one batch

to use the Teuchos memory management types, even if there are good unit tests in place. Refactorings should *never* be attempted in large batches, no matter what [15].

The above process was followed with great success to refactor the Trilinos package Thyra [3] over a period of more than a year. This process can be followed for a code base very safely and productively if the above incremental unit-testing refactoring process is followed.

# 6 Miscellaneous topics

When thinking about memory management in C++ it is helpful to take a step back and consider a few different higher-level issues. In the following section, the issue of essential and accidental complexity is discussed and what role the Teuchos memory management classes play in addressing accidental complexity and helping to make implicit concepts explicit. Then, the philosophy of memory management is discussed and some analogies are used to help put things in perspective and provide a solid foundation for the approach used in the Teuchos memory management classes as compared to approaches that start with safer language but arrive at a similar balance between safety, speed, and flexibility.

## 6.1 Essential and accidental complexity, making implicit concepts explicit

While the idioms described in this document (largely outlined in Section 5.8) may appear complex at first sight, one has to consider that it is not really the idioms that are complex but the essential attributes of object relationships that are complex. Frederick Brooks refers to this as *essential complexity* as opposed to *accidental complexity* [8]. *Accidental complexity* in programming refers to complexity resulting from complicating details of the programming language or environment which are not directly related to solving the problem at hand. Accidental complexity has largely been removed as higher level languages have been developed [8, Chapter 16]. However, raw pointers in C and C++ and manual resource management (when that is not the main focus of the program) are definitely a lingering category of accidental complexity[21]. Alternatively, *essential complexity* exists because of the nature of the problem at hand that no programming language will ever be able to fully remove. (However, one can use object-oriented and other design approaches to abstract and partition the essential complexity such that we can write and maintain complex large-scale programs.)

What the Teuchos Memory Management classes do is that they remove much of the accidental complexity of using raw pointers and manual resource management and instead they more directly address the essential complexity of writing programs in making important concepts of object relationships explicit that are instead implicit in most languages (including raw C++). Dealing with the nature of relationships between objects is essential complexity and for every relationship between two classes (for example in a UML class diagram [16]) one must answer the essential questions:

- *What is the multiplicity of the relationship?* (i.e. is there just one object or is there more than one object at the other end of the association?). In UML class diagrams, a singular multiplicity relationship is represented using a `1` and multiplicity greater than one is represented using `1..*` (see Figure 4 for examples).
- *Is the object optional or required?* In a UML class diagram, an optional object is represented using `0..1` while a required object is represented as `1` (see Figure 4 for examples).
- *Is the object changeable or non-changeable?* In UML class diagrams, a non-changeable object is given the attribute {`readOnly`}. In UML, by default, all objects at the end of an association are assumed changeable.
- *Is the association persisting or non-persisting?* In a UML class diagram, non-persisting associations are referred to as "dependency associations" and are represented with a dotted line (and can also be

[21] `http://discuss.joelonsoftware.com/default.asp?joel.3.278613.51`

given the keyword ≪`parameter`≫). Persisting associations are referred to as "relationships" and are represented as solid lines (see Figure 4 for examples).

Note that while UML is an expressive language that allows one to explicitly represent the above essential information, most programming languages cannot (at least not the raw language). Consider that in Java and Python that it is impossible to distinguish between persisting and non-persisting associations because every user-defined object is always managed through an indirect reference handed by the garbage-collected language. This causes big problems when it comes time to try to understand a complex program written in these languages. For example, consider the agony that Micheal Feathers goes through in many refactorings described in [15] in trying to determine the nature of objects as to whether they are actually embedded in each other (i.e. persisting) or are just passed to each other (i.e. non-persisting). Python has no user-definable concept of `const` but the Python language itself understands the need for `const` by having built-in immutable data-types like strings and tuples.

One of the goals of the idioms defined in this paper is to change the above essential complexities from implicit concepts to explicit concepts directly stated in code (see "Making Implicit Concepts Explicit" in [14, Chapter 9]). The essential attributes of object relationships (i.e. multiplicity, persistent vs. non-persistent, changeable vs. non-changeable) are present in every program no mater what high-level programming language is used [8, 24, 15]. The issue is that most executable languages (not withstanding executable XML [16, Chapter 1]) lack the expressiveness to make these concepts explicit. The Teuchos memory management classes and the associated idioms described in this paper provide a means to make many of these essential concepts explicit in C++ in a way that is not possible in any other widely used programming language.

While the Teuchos memory management classes go a long way in removing some of the accidental complexity of programing in C++ due to manual memory management, some of the types of remaining accidental complexity (among many others not mentioned) include:

- *Value semantics versus reference semantics*: The distinction between value semantics versus reference semantics is a C++ concept that does not directly relate to solving a problem or representing a model in code and is therefore accidental complexity. In Java and Python, all user-defined types use reference semantics but in C++ programmers can take advantage of value-types which gives more efficient code and more control in C++ than what is possible in these other languages. However, this extra control could be classified as accidental complexity (which we tolerate for the sake of added control and improved performance).
- *Pointer syntax for memory management types `Ptr`, and `RCP`*: In order to access the underlying object through the smart pointer types `Ptr` and `RCP`, one has to use pointer syntax using `func(*a)` and `a->someMember()`. The C++ language makes it impossible to define abstract data types that allow direct access to the underlying object (i.e. using `func(a)` and `a.someMember()`) like raw C++ references allow. Pointer syntax is not essential to the nature of problem solving (as proven by all the languages that don't have pointers including Java and Python) and therefore pointer syntax must be categorized as accidental complexity. Note, however, that the types `ArrayView` and `ArrayRCP` were not listed in this category because one can use these array classes using just the `operator[](size_type)` function and one does not need to use pointer syntax. In fact, `ArrayView` does not even support any pointer-like functions and the pointer-like functions on `ArrayRCP` are only really there to allow it to be used as a general-purpose checked iterator in a debug-mode build. While pointer syntax is not an essential concept they do actually come in handy to define iterators into

containers and present a much more compact iterator interface that what one will find in other languages. In other words, pointers syntax can be considered to be a nice enhancement when considering iterators. In most other contexts, however, we must consider pointer syntax to contribute to accidental complexity.

## 6.2 Philosophy of memory management: Safety, speed, flexibility and 100% guarantees

When looking at different strategies for memory management in C++ and in other languages, it helps to think a little on the philosophical level which can actually help put the issues involved in perspective.

When looking at the different memory management approaches implemented in various programming languages, the core issues come down to trade-offs in safety and correctness versus speed and flexibility. For example, a language like C sacrifices safety and correctness for speed and flexibility. Because C is so "close" to the hardware, one can implement very specialized memory management approaches tailored to very specific types of domains. However, the price one pays for this raw speed and flexibility in C is the fact that there is very little compiler-supported checking that would otherwise be needed to assert correct memory usage.

Now take Python on the other extreme. If one writes code only in Python, one will almost never experience and memory leak or segfault of any kind due to code that one directly writes. Here is a language which is nearly 100% safe (assuming the language implementation is 100% correct) but offers less flexibility in how memory is managed and results in very slow native code as compared to C in many cases (e.g. for computationally intensive loops).

So how important is a 100% guarantee that memory will always be used correctly like is provided in a language like Python? How important is a 100% guarantee in any area? Well, if one can get a 100% guarantee without having to pay a significant price for it then one would be a fool not to accept it. For example, if one has a choice between two vendors selling the same product for the same price but one vendor will give a 100% money-back guarantee, with all things being equal, it would probably be foolish not to go with the vendor with the 100% guarantee.

However, in most areas, greater safety (not to mention a 100% guarantee) comes with greater costs. Instead of demanding a 100% guarantee, we typically accept some level of extra risk as long as we have taken basic precautions to protect ourselves. To demonstrate this, let's consider another analogy which I like to refer to as the *Transportation Analogy*. When considering modes of transportation, we accept that we are not 100% safe when driving our cars on the road but we do it anyway. The reason that we get into our cars every day is that we take reasonable precautions like purchasing a car with a good safely design, wearing seat-belts, obeying the traffic laws, driving a reasonable speed, and practicing defensive driving. What is going to be argued here is that the approach to memory management that is being advocated in this paper is the equivalent of driving a car, wearing one's seat belt, and taking other reasonable safety precautions but does not provide a 100% guarantee.

Now let's talk about the safety versus speed/efficiently extremes in the Transportation Analogy and in the area of memory management. At one extreme, writing all high-level code in C++ (or C) using raw pointers for everything is like riding a high-performance motorcycle on a crowded interstate going 150 mph, without wearing a helmet or any other safety gear, while doing a wheelie. At this extreme, one wrong move means certain death.

At the other extreme, writing all code in a language like Python is like driving around in a reinforced tank that does a maximum of 10 mph where one sits inside wearing a car racing helmet with the Hans device, full racing safety gear, and having a massive air bag system to encase one's entire body in foam three feet thick in case of a collision. On this side of extreme safely, we could hit a Mac truck head on and be just fine. The only way to really kill one's self would be to drive off a shear cliff.

If we all required a near 100% safely guarantee, we would all be driving around in reinforced tanks like the one described above but we don't. We don't because we are not willing to pay the price of the near 100% guarantee provided by the tank. We can't afford it financially and it would take forever to get back and forth to work. Instead, we are content with our less than 100% safe cars because they are affordable and fast and yet do not pose unreasonable risks.

Now, we can incrementally go from either extreme to a more balanced state in both the Transportation Analogy and with memory management in C++ and Python.

From the extreme of safety with less speed and flexibility represented by the reinforced tank (and Python) one can incrementally move toward the middle ground of the car. One can start by removing the racing helmet and Hans device, followed by decreasing the weight and increasing the speed of the tank, and so on. Continuing on this trend of sacrificing safety in favor of greater speed and agility leads us to our typical car. Likewise, moving from an extreme of safety to a more reasonable balance between safety and speed/flexibility in Python involves taking pieces of computationally intensive Python code, rewriting them in C/C++, and then calling them from Python. This is an approach that many Python enthusiasts are advocating [22] but make no mistake that in going down this road that one is sacrificing safety in Python in the name of speed and flexibility. One is giving up Python's near 100% guarantee when one does this and will therefore have to deal with difficult and dangerous memory errors cropping up in the code.

From the extreme of speed and flexibility with little regard for safety represented by the motorcycle (and C/C++ raw pointers) one can also incrementally move toward the car. One can start by putting on a helmet, followed by slowing down some, and so on. Continuing on this trend of adding safety will eventually see one morphing the motorcycle into the typical car. Likewise, moving from an extreme of less safety toward a more reasonable balance between speed/flexibility and safety in C++ involves adding more and more utility classes to hide more and more uses of raw C++ pointers in high-level C++ code. This is the trend that the C++ community has been following for more than the last decade. We see it first in the introduction of `std::auto_ptr` and `std::vector`. This was then followed by the development of `boost::shared_ptr` (and therefore `std::shared_ptr` in C++0x). What is being suggested in this paper is the logical conclusion of this journey which is the development of a complete set of utility classes in order to remove all raw C++ pointers from high-level C++ code; i.e. complete the transition from the motorcycle (C/C++ raw pointers) to the car (Teuchos C++ memory management classes).

With the approach being advocated in this paper, using the Teuchos memory management classes in debug-mode is like driving around in the tank where one is protected from almost any danger. However, using the Teuchos memory management classes in non-debug optimized builds is like driving around with the fast high-performance motorcycle. Try turning a tank into a motorcycle and then back again that easily!

# 7 Conclusions

Using the Teuchos reference-counted memory management classes allows one to remove unnecessary constraints in the use of objects by removing arbitrary lifetime ordering constraints which are a type of unnecessary coupling [24]. The code one writes with these classes will be more likely to be correct on first writing, will be less likely to contain silent (but deadly) memory usage errors, and will be much more robust to later refactoring and maintenance.

The level of debug-mode runtime checking provided by the Teuchos memory management classes is stronger in many respects than what is provided by memory checking tools like Valgrind and Purify while being much less expensive. However, tools like Valgrind and Purify perform a number of types of checks (like usage of uninitialized memory) that makes these tools very valuable and therefore complement the Teuchos memory management debug-mode runtime checking.

The Teuchos memory management classes and idioms largely address the technical issues in resolving the fragile built-in C++ memory management model (with the exception of circular references which has no easy solution but can be managed as discussed). All that remains is to teach these classes and idioms and expand their usage in C++ codes. The long-term viability of C++ as a usable and productive language depends on it. Otherwise, if C++ is no safer than C, then is the greater complexity of C++ worth what one gets as extra features? Given that C is smaller and easier to learn than C++ and since most programmers don't know object-orientation (or templates or X, Y, and Z features of C++) all that well anyway, then what really are most programmers getting extra out of C++ that would outweigh the extra complexity of C++ over C? C++ zealots will argue this point but the reality is that C++ popularity has peaked and is becoming less popular while the popularity of C has remained fairly stable over the last decade[22]. Idioms like are advocated in this paper can help to avert this trend but it will require wide community buy-in and a change in the way C++ is taught in order to have the greatest impact.

To make these programs more secure, compiler vendors or static analysis tools (e.g. klocwork[23]) could implement a preprocessor-like language similar to OpenMP[24] that would allow the programmer to declare (in comments) that certain blocks of code should be "pointer-free" or allow smaller blocks to be "pointers allowed". This would significantly improve the robustness of code that uses the memory management classes described here.

[22] See the Tiobe index of programming language popularity at `http://www.tiobe.com`.
[23] `http://www.klocwork.com`
[24] `http://openmp.org`

# References


[1] A. Avram and F. Marinescu. *Domain-Driven Design Quickly*. InfoQ, 2006.

[2] R. A. Bartlett. Teuchos::RCP : An introduction to the Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++. Technical report SAND04-3268, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2004.

[3] R. A. Bartlett. Thyra linear operators and vectors: Overview of interfaces and support software for the development and interoperability of abstract numerical algorithms. Technical report SAND2007-5984, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2007.

[4] R. A. Bartlett. Mathematical and high-level overview of MOOCHO: The Multifunctional Object-Oriented arCHitecture for Optimization. Technical report SAND09-3969, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2009.

[5] R. A. Bartlett. Thyra coding and documentation guidelines (tcdg) version 1.0. Technical report SAND2010-2051, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2010.

[6] R. A. Bartlett, B. G. van Bloeman Waanders, and M. A. Heroux. Vector reduction/transformation operators for linear algebra interfaces to efficiently develop complex abstract numerical algorithms independently of data mapping, 2003. Submitted to *ACM TOMS*.

[7] BOOST. The BOOST library. http://www.boost.org.

[8] F. Brooks. *The Mythical Man-Month (second edition)*. Addison Wesley, 1995.

[9] J. Coplien. *Advanced C++*. Addison Wesley, 1992.

[10] Micosoft Corporation. COM: Component object model. http://www.microsoft.com/com.

[11] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[12] S. Dewhurst. *C++ Gotchas*. Addison Wesley, 2003.

[13] Glenn Downing, Paul F. Dubois, and Teresa Cottom. Data sharing in scientific simulations. *Computing in Science and Engineering*, 6:87–96, 2004.

[14] E. Evans. *Domain-Driven Design*. Addison Wesley, 2004.

[15] M. Feathers. *Working Effectively with Legacy Code*. Addison Wesley, 2005.

[16] M. Fowler. *UML Distilled (third edition)*. Addison Wesley, 2004.

[17] E. Gamma et al. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[19] Object Management Group. CORBA: Common object request broker architecture. http://www.corba.org.

[20] M. Heroux. Design issues for numerical libraries on scalable multicore architectures. *Journal of Physics: Conference Series*, 2008.

[21] P. Kambadur1, D. Gregor1, A. Lumsdaine, and A. Dharurkar. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, chapter Modernizing the C++ Interface to MPI, pages 266–274. Springer, 2006.

[22] H. Langtengen and X. Cai. On the efficency of python for high-performance computing: A case study involving stencil updates for partial differential equations. In H. Bock, E. Kostina, X. Hoang, and R. Rannacher, editors, *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the Third International Conference on High Performance Scientific Computing*, pages 337–358. Springer-Varlag Berlin Heidelberg, 2008.

[23] R. Martin. *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall, 2003.

[24] S. McConnell. *Code Complete: Second Edition*. Microsoft Press, 2004.

[25] Scott Meyers. *More Effective C++*. Addison Wesley, 1996.

[26] Scott Meyers. *Effective C++ (Third Edition)*. Addison Wesley, 2005.

[27] E. Phipps, R. Bartlett, and D. Gay. Automatic differentiation of C++ codes for large-scale scientific computing. *Third International Workshop on Automatic Differentiation*, February 2006.

[28] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, New York, 1994.

[29] B. Stroustrup. *The C++ Programming Language, special edition*. Addison-Wesley, New York, 2000.

[30] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1 to 4–59, 2007.

[31] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines and Best Practices*. Addison Wesley, 2005.

[32] Bart van Bloemen Waanders, Roscoe Bartlett, Kevin Long, Paul Boggs, and Andrew Salinger. Large scale non-linear programming for PDE constrained optimization. Technical report SAND2002-3198, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2002.

[33] M. VanDerVanter, D.E. Post, and M.E. Zosel. HPC needs a tool strategy. Technical report LA-UR-05-1592, Las Alamos Laboratories, 2005.

# A Summary of Teuchos memory management classes and idioms

**Basic Teuchos smart pointer types**

| | Non-persisting (and semi-persisting) Associations | Persisting Associations |
|---|---|---|
| single objects | `Ptr<T>` | `RCP<T>` |
| contiguous arrays | `ArrayView<T>` | `ArrayRCP<T>` |

**Other Teuchos array container classes**

| Array class | Specific use case |
|---|---|
| `Array<T>` | Contiguous dynamically sizable, expandable, and contractible arrays |
| `Tuple<T,N>` | Contiguous statically sized (with size `N`) arrays |

**Equivalencies for const protection for raw pointers and Teuchos smart pointers types**

| Description | Raw pointer | Smart pointer |
|---|---|---|
| Basic declaration (non-const obj) | `typedef A* ptr_A` | `RCP<A>` |
| Basic declaration (const obj) | `typedef const A* ptr_const_A` | `RCP<const A>` |
| non-const pointer, non-const object | `ptr_A` | `RCP<A>` |
| const pointer, non-const object | `const ptr_A` | `const RCP<A>` |
| non-const pointer, const object | `ptr_const_A` | `RCP<const A>` |
| const pointer, const object | `const ptr_const_A` | `const RCP<const A>` |

**Summary of operations supported by the basic Teuchos smart pointer types**

| Operation | `Ptr<T>` | `RCP<T>` | `ArrayView<T>` | `ArrayRCP<T>` |
|---|---|---|---|---|
| *Raw pointer-like functionality* | | | | |
| Implicit conv derived to base | x | x | | |
| Implicit conv non-const to const | x | x | x | x |
| Dereference `operator*()` | x | x | | x |
| Member access `operator->()` | x | x | | x |
| `operator[](i)` | | | x | x |
| operators `++`, `--`, `+=(i)`, `-=(i)` | | | | x |
| *Other functionality* | | | | |
| Reference counting machinery | | x | | x |
| Iterators: begin(), end() | | | x | x |
| `ArrayView` subviews | | | x | x |

**Basic implicit and explicit supported conversions for Teuchos smart pointer types**

| Operation | `Ptr<T>` | `RCP<T>` | `ArrayView<T>` | `ArrayRCP<T>` |
|---|---|---|---|---|
| Implicit conv derived to base | x | x | | |
| Implicit conv non-const to const | x | x | x | x |
| `const_cast` | x | x | x | x |
| `static_cast` | x | x | | |
| `dynamic_cast` | x | x | | |
| `reinterpret_cast` | | | x | x |

**Class Data Members for Value-Type Objects**

| Data member purpose | Data member declaration |
|---|---|
| non-shared, single, const object | `const S s_;` |
| non-shared, single, non-const object | `S s_;` |
| non-shared array of non-const objects | `Array<S> as_;` |
| shared array of non-const objects | `RCP<Array<S> > as_;` |
| non-shared statically sized array of non-const objects | `Tuple<S,N> as_;` |
| shared statically sized array of non-const objects | `RCP<Tuple<S,N> > as_;` |
| shared fixed-sized array of const objects | `ArrayRCP<const S> as_;` |
| shared fixed-sized array of non-const objects | `ArrayRCP<S> as_;` |

**Class Data Members for Reference-Type Objects**

| Data member purpose | Data member declaration |
|---|---|
| non-shared or shared, single, const object | `RCP<const A> a_;` |
| non-shared or shared, single, non-const object | `RCP<A> a_;` |
| non-shared array of const objects | `Array<RCP<const A> > aa_;` |
| non-shared array of non-const objects | `Array<RCP<A> > aa_;` |
| shared fixed-sized array of const objects | `ArrayRCP<RCP<const A> > aa_;` |
| "..." (const ptr) | `ArrayRCP<const RCP<const A> > aa_;` |
| shared fixed-sized array of non-const objects | `ArrayRCP<RCP<const A> > aa_;` |
| "..." (const ptr) | `ArrayRCP<const RCP<const A> > aa_;` |

**Passing IN Non-Persisting Associations to Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object (required) | `const A &a` |
| single, non-changeable object (optional) | `const Ptr<const A> &a` |
| single, changeable object (required) | `const Ptr<A> &a` **or** `A &a` |
| single, changeable object (optional) | `const Ptr<A> &a` |
| array of non-changeable objects | `const ArrayView<const Ptr<const A> > &aa` |
| array of changeable objects | `const ArrayView<const Ptr<A> > &aa` |

**Passing IN Persisting Associations to Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const RCP<const A> &a` |
| single, changeable object | `const RCP<A> &a` |
| array of non-changeable objects | `const ArrayView<const RCP<const A> > &aa` |
| array of changeable objects | `const ArrayView<const RCP<A> > &aa` |

**Passing OUT Persisting Associations for Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const Ptr<RCP<const A> > &a` |
| single, changeable object | `const Ptr<RCP<A> > &a` |
| array of non-changeable objects | `const ArrayView<RCP<const A> > &aa` |
| array of changeable objects | `const ArrayView<RCP<A> > &aa` |

**Passing OUT Semi-Persisting Associations for Reference (or Value) Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object | `const Ptr<Ptr<const A> > &a` |
| single, changeable object | `const Ptr<Ptr<A> > &a` |
| array of non-changeable objects | `const ArrayView<Ptr<const A> > &aa` |
| array of changeable objects | `const ArrayView<Ptr<A> > &aa` |

**Passing IN Non-Persisting Associations to Value Objects as Func Args**

| Argument Purpose | Formal Argument Declaration |
|---|---|
| single, non-changeable object (required) | `S s` **or** `const S s` **or** `const S &s` |
| single, non-changeable object (optional) | `const Ptr<const S> &s` |
| single, changeable object (required) | `const Ptr<S> &s` **or** `S &s` |
| single, changeable object (optional) | `const Ptr<S> &s` |
| array of non-changeable objects | `const ArrayView<const S> &as` |
| array of changeable objects | `const ArrayView<S> &as` |

**Passing IN Persisting Associations to Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const ArrayRCP<const S> &as` |
| array of changeable objects | `const ArrayRCP<S> &ss` |

**Passing OUT Persisting Associations for Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const Ptr<ArrayRCP<const S> > &as` |
| array of changeable objects | `const Ptr<ArrayRCP<S> > &as` |

**Passing OUT Semi-Persisting Associations for Value Objects as Func Args**

(Use cases not covered by reference semantics used for value types)

| Argument Purpose | Formal Argument Declaration |
|---|---|
| array of non-changeable objects | `const Ptr<ArrayView<const S> > &as` |
| array of changeable objects | `const Ptr<ArrayView<S> > &as` |

**Returning Non-Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single cloned object | `RCP<A>` |
| Single non-changeable object (required) | `const A&` |
| Single non-changeable object (optional) | `Ptr<const A>` |
| Single changeable object (required) | `A&` |
| Single changeable object (optional) | `Ptr<A>` |
| Array of non-changeable objects | `ArrayView<const Ptr<const A> >` |
| Array of changeable objects | `ArrayView<const Ptr<A> >` |

**Returning Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single non-changeable object | `RCP<const A>` |
| Single changeable object | `RCP<A>` |
| Array of non-changeable objects | `ArrayView<const RCP<const A> >` |
| Array of changeable objects | `ArrayView<const RCP<A> >` |

**Returning Semi-Persisting Associations to Reference (or Value) Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single non-changeable object | `Ptr<const A>` |
| Single changeable object | `Ptr<A>` |
| Array of non-changeable objects | `ArrayView<const Ptr<const A> >` |
| Array of changeable objects | `ArrayView<const Ptr<A> >` |

**Returning Non-Persisting Associations to Value Objects**

| Purpose | Return Type Declaration |
|---|---|
| Single copied object (return by value) | `S` |
| Single non-changeable object (required) | `const S&` |
| Single non-changeable object (optional) | `Ptr<const S>` |
| Single changeable object (required) | `S&` |
| Single changeable object (optional) | `Ptr<S>` |
| Array of non-changeable objects | `ArrayView<const S>` |
| Array of changeable objects | `ArrayView<S>` |

**Returning Persisting Associations to Value Objects**

(Use cases not covered by reference semantics used for value types)

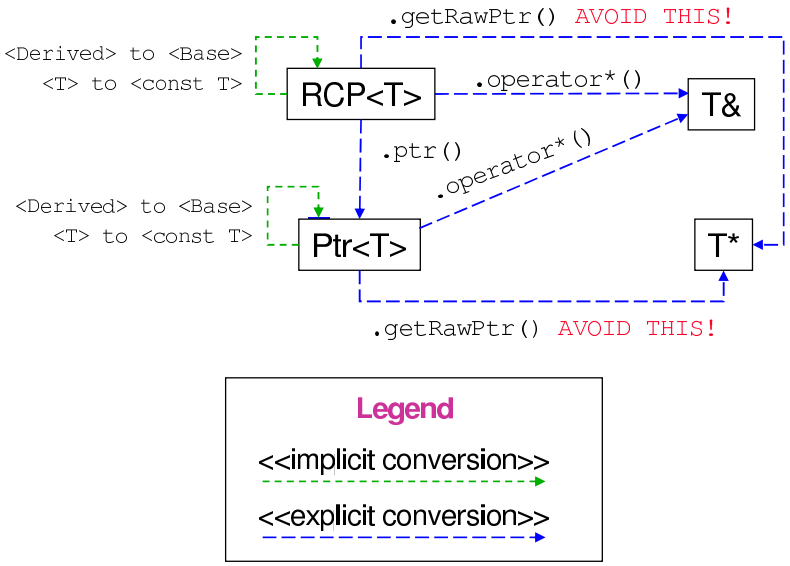
| Purpose | Return Type Declaration |
|---|---|
| Array of non-changeable objects | `ArrayRCP<const S>` |
| Array of changeable objects | `ArrayRCP<S>` |

**Returning Semi-Persisting Associations to Value Objects**

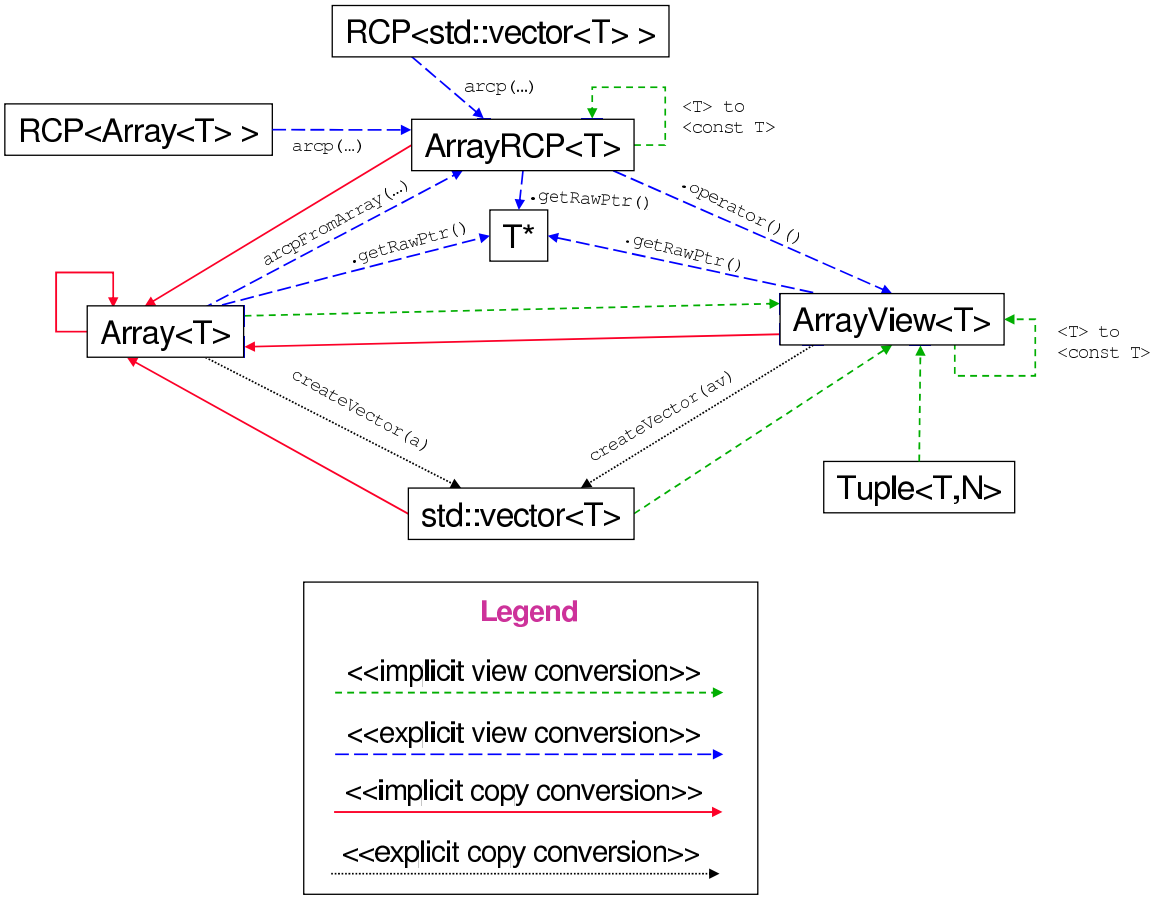
(Use cases not covered by reference semantics used for value types)

| Purpose | Return Type Declaration |
|---|---|
| Array of non-changeable objects | `ArrayView<const S>` |
| Array of changeable objects | `ArrayView<S>` |

**Conversions of data-types for single objects**




**Conversions of data-types for contiguous arrays**



**Most Common Basic Conversions for Single Object Types**

| Type To | Type From | Properties | C++ code |
|---|---|---|---|
| RCP<A> | A* | Ex, Ow | rcp(a_p) [1] |
| RCP<A> | A* | Ex, NOw | rcp(a_p,false) [2] |
| RCP<A> | A& | Ex, NOw | rcpFromRef(a) |
| RCP<A> | A& | Ex, NOw | rcpFromUndefRef(a) |
| RCP<A> | Ptr<A> | Ex, NOw, DR | rcpFromPtr(a) |
| RCP<A> | boost::shared_ptr<A> | Ex, Ow, DR | rcp(a_sp) |
| RCP<const A> | RCP<A> | Im, Ow, DR | RCP<const A>(a_rcp) |
| RCP<Base> | RCP<Derived> | Im, Ow, DR | RCP<Base>(derived_rcp) |
| RCP<const Base> | RCP<Derived> | Im, Ow, DR | RCP<const Base>(derived_rcp) |
| boost::shared_ptr<A> | RCP<A> | Ex, Ow, DR | shared_pointer(a_rcp) |
| A* | RCP<A> | Ex, NOw | a_rcp.getRawPtr() [3] |
| A& | RCP<A> | Ex, NOw | *a_rcp [4] |
| Ptr<A> | A* | Ex, NOw | ptr(a_p) [2] |
| Ptr<A> | A& | Ex, NOw | outArg(a) [5] |
| Ptr<A> | RCP<A> | Ex, NOw, DR | a_rcp.ptr() |
| Ptr<A> | RCP<A> | Ex, NOw, DR | a_rcp() |
| Ptr<A> | RCP<A> | Ex, NOw, DR | ptrFromRCP(a_rcp) |
| Ptr<const A> | Ptr<A> | Im, NOw, DR | Ptr<const A>(a_ptr) |
| Ptr<Base> | Ptr<Derived> | Im, NOw, DR | Ptr<Base>(derived_ptr) |
| Ptr<const Base> | Ptr<Derived> | Im, NOw, DR | Ptr<const Base>(derived_ptr) |
| A* | Ptr<A> | Ex, NOw | a_ptr.getRawPtr() [3] |
| A& | Ptr<A> | Ex, NOw | *a_ptr() [4] |
| A* | A& | Ex, NOw | &a [3] |
| A& | A* | Ex, NOw | *a_p [3] |

Types/identifiers: `A* a_p`; `A& a`; `Ptr<A> a_ptr`; `RCP<A> a_rcp`; `boost::shared_ptr<A> a_sp`;

Properties: Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning, NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection (NOTE: All conversions are shallow conversions, i.e. copies pointers not objects.)

1. Constructing an owning `RCP` from a raw C++ pointer is strictly necessary but must be done with great care according to the commandments in Appendix B.
2. Constructing a non-owning `RCP` or `Ptr` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer and raw pointer manipulation should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section 5.2).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.
5. See other helper constructors for passing `Ptr` described in Section 5.4.1.

**Most Common Basic Conversions for Contiguous Array Types**

| Type To | Type From | Properties | C++ code (or impl function) |
|---|---|---|---|
| ArrayRCP<S> | S* | Sh, Ex, Ow | arcp(s_p,0,n) [1] |
| ArrayRCP<S> | S* | Sh, Ex, NOw | arcp(s_p,0,n,false) [2] |
| ArrayRCP<S> | Array<S> | Sh, Ex, NOw, DR | arcpFromArray(s_a) |
| ArrayRCP<S> | ArrayView<S> | Sh, Ex, NOw, DR | arcpFromArrayView(s_av) |
| ArrayRCP<S> | ArrayView<S> | Dp, Ex, Ow | arcpClone(s_av) |
| ArrayRCP<S> | RCP<Array<S> > | Sh, Ex, Ow, DR | arcp(s_a_rcp) |
| ArrayRCP<const S> | RCP<const Array<S> > | Sh, Ex, Ow, DR | arcp(cs_a_rcp) |
| ArrayRCP<const S> | ArrayRCP<S> | Sh, Im, Ow, DR | ArrayRCP::operator()() |
| S* | ArrayRCP<S> | Sh, Ex, NOw | s_arcp.getRawPtr() [3] |
| S& | ArrayRCP<S> | Sh, Ex, NOw | s_arcp[i] [4] |
| ArrayView<S> | S* | Sh, Ex, NOw | arrayView(s_p,n) [1] |
| ArrayView<S> | Array<S> | Sh, Im, NOw, DR | Array::operator ArrayView() |
| ArrayView<S> | Tuple<S> | Sh, Im, NOw, DR | Tuple::operator ArrayView() |
| ArrayView<S> | std::vector<S> | Sh, Im, NOw | ArrayView<S>(s_v) |
| ArrayView<S> | ArrayRCP<S> | Sh, Ex, NOw, DR | ArrayRCP::operator()() |
| ArrayView<const S> | const Array<S> | Sh, Im, NOw, DR | Array::operator ArrayView() |
| ArrayView<const S> | const Tuple<S> | Sh, Im, NOw, DR | Tuple::operator ArrayView() |
| ArrayView<const S> | const std::vector<S> | Sh, Im, NOw | ArrayView(cs_v) |
| ArrayView<const S> | ArrayRCP<const S> | Sh, Ex, NOw, DR | ArrayRCP::operator ArrayView() |
| S* | ArrayView<S> | Ex, NOw | s_av.getRawPtr() [3] |
| S& | ArrayView<S> | Ex, NOw | s_av[i] [4] |
| Array<S> | S* | Dp, Ex | Array<S>(s_p,s_p+n) |
| Array<S> | std::vector<S> | Dp, Im | Array<S>(s_v) |
| Array<S> | ArrayView<S> | Dp, Im | Array<S>(s_av) |
| Array<S> | Tuple<S,N> | Dp, Im | Array<S>(s_t) |
| Array<S> | ArrayRCP<S> | Dp, Ex | Array<S>(s_arcp()); |
| std::vector<S> | Array<S> | Dp, Ex | s_a.toVector(); |
| S* | Array<S> | Ex, NOw | s_a.getRawPtr() [3] |
| S& | Array<S> | Ex, NOw | s_a[i] [4] |

Types/identifiers: `S* s_p; ArrayView<S> s_av; ArrayRCP<S> s_arcp; Array<S> s_a; Tuple<S,N> s_t; std::vector<S> s_v; RCP<Array<S> > s_a_rcp; RCP<const Array<S> > cs_a_rcp;`

Properties: Sh = Shallow copy, Dp = Deep copy (dangling references not an issue), Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning (dangling references not an issue), NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection for non-owning

1. It should never be necessary to convert from a raw pointer to an owning `ArrayRCP` object directly. Instead, use the non-member constructor `arcp<S>(n)`.
2. Constructing a non-owning `ArrayRCP` or `ArrayView` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section 5.2).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.

# B Commandments for the use of the Teuchos memory management classes

Here are stated commandments (i.e. very strongly recommended guidelines) that if followed, along with the idioms defined in Section 5.8, then client code will be nearly 100% safe through debug-mode runtime checking and will almost never result in undefined behavior (e.g. segfaults) or a memory leak (except for circular references as described in Section 5.11.2). While there will be situations where it is justified to violate almost any of these commandments, they should be valid in 99% of a well written code base.

**Commandment 1** *Thou shall not expose raw pointers in any high-level C++ code.*

*Exception:* Only expose raw pointers when interfacing with non-compliant code or momentarily in order to construct a Teuchos memory management class object. However, these cases should be encapsulated as low-level code.

**Commandment 2** *Thou shall only use raw C++ references for non-persisting associations (see Sections 4.2 and 5.4.3).*

**Commandment 3** *Thou shall use `RCP` for handling single objects for all persisting associations (see Section 4.2).*

**Commandment 4** *Thou shall put a pointer for an object allocated with operator `new` into a strong owning `RCP` whenever possible by directly calling `new` right in the constructor for the `RCP` object itself or construct from `rcp(...)`.*

**Commandment 5** *When wrapping an object inside of an `RCP`, thou shall create a strong owning `RCP` object first before any non-owning `RCP` objects (see Sections 5.11.4 and 5.13.3).*

*Justification:* In order for the reference-counting machinery to detect dangling non-owning references in a debug-mode build, the first `RCP` object created must have ownership to delete. The system cannot detect dangling references from non-owning `RCPNode` objects created before an owning `RCPNode` object is created.

**Commandment 6** *Thou shall use `Ptr` for handling single objects for all semi-persisting associations (see Section 4.2).*

*Justification:* When performance constraints do not allow for the reference-counting overhead of `RCP`, then `Ptr` can be used instead to form a semi-persisting association (which should be accompanied with the appropriate documentation about the performance optimization). One should never have to retreat back to using a raw pointer in these cases. At least with `Ptr`, invalid usage will be checked for in a debug build so one does not loose any debug-mode runtime checking when using `Ptr` instead of `RCP` if one really does not need reference-counting machinery.

**Commandment 7** *Thou shall use* `ArrayRCP` *for handling all contiguous arrays of objects for all persisting associations where the array does not need to be incrementally resized while sharing the array (see Sections 4.2 and 5.8.2).*

**Commandment 8** *Thou shall use* `ArrayView` *for handling contiguous arrays of objects for all semi-persisting associations (see Sections 4.2 and 5.12.3).*

*Justification:* When performance constraints do not allow for the reference-counting overhead of `ArrayRCP`, then `ArrayView` can be used instead to form a semi-persisting association (which should be accompanied with the appropriate documentation about the performance optimization). One should never have to retreat back to using a raw pointer in these cases. At least with `ArrayView`, invalid usage will be checked for in a debug build so one does not loose any debug-mode runtime checking when using `ArrayView` instead of `ArrayRCP` if one really does not need reference-counting machinery.

**Commandment 9** *Thou shall not call raw* `new` *or* `delete` *in any high-level C++ code to dynamically allocate and destroy single objects. Instead, create memory using a user-defined non-member constructor function (see Section 5.8.1).*

*Exception*: Calling raw `new` in okay when an appropriate non-member constructor is missing. In general, value-type classes (e.g. `std::vector`) will not have non-member constructor functions that return `RCP`-wrapped objects.

**Commandment 10** *Thou shall not call raw operator* `new []` *or* `delete []` *in any high-level C++ code to dynamically allocate and destroy contiguous arrays of data. Instead, use functions such as* `Teuchos::Array<T>(n)` *and* `Teuchos::arcp<T>(n)` *to dynamically allocate arrays.*

**Commandment 11** *Thou shall not directly create and use compile-time fixed sized arrays with* `T[N]`*. Instead, create compile-time fixed-sized arrays using* `Teuchos::Tuple<T,N>` *and convert to* `Teuchos::ArrayView<T>` *for more general usage.*

**Commandment 12** *Thou shall use* `Teuchos::Array` *as a general purpose contiguous container instead of* `std::vector` *in order maximize debug-mode runtime checking (see Sections 5.5.3 and 5.11.3).*

**Commandment 13** *Thou shall only convert or cast between different memory management objects (of the same or different types) using the provided implicit and explicit conversion functions (see Section 5.7). Thou shall never expose a raw C++ pointer to perform a conversion.*

*Exception*: Some very advanced and rare use cases might have one exposing a raw C++ pointer (see Section 5.13.1 for the only example described in this paper).

**Commandment 14** *Thou shall only pass in the types* `Ptr`*,* `RCP`*,* `ArrayView`*, and* `ArrayRCP` *by constant reference (e.g.* `const RCP<T> &a`*) and never by non-const reference (e.g. never do* `RCP<T> &a`*).*

*Exception*: The only time one should ever pass in a non-const reference to one of these types (e.g. `RCP<T> &a`) is when the function will modify what data the object points to. However, if this is the case, it is typically better and more clear to pass in the object through a `Ptr` object (e.g. `const Ptr<RCP<T> > &a`) using the `outArg(...)` function (see Section 5.8.4).

**Commandment 15** *Thou shall only reutrn objects of type `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` from a function by value and not a constant reference (see Section 5.8.5).*

*Exception*: Returning one of these types by non-const reference makes sense when using the local static variable initialization trick described in [26, Item 4]. However, returning one of these types by const reference would almost never be justified.

# C Argument for using a signed integer for `size_type` in the Teuchos array classes

The Teuchos array memory management classes `Array`, `ArrayRCP`, and `ArrayView` all use a signed integer for `size_type` (`ptrdiff_t` by default). This breaks from the C++ standard library convention of the standard containers like `std::vector` that all use an unsigned integer for `size_type` (which is `size_t` in most implementations). The primary disadvantage for using an unsigned integral type is that subtractions that would normally produce a negative number instead roll over into a huge positive number, making it more difficult to debug problems. For example, consider the simple program shown in Listing 127:

**Listing 127** *: Example program showing the problem with unsigned integral types*

```
#include <iostream>
#include <string>

typedef unsigned long int size_type ;

void print_val(const std::string &valName, const size_type val)
{ std::cout << valName << " = " << val << "\n";}

int main()
{
  const size_type a = 5, b = 7;
  const size_type c = b - a;
  const size_type d = a - b;
  print_val("a", a);
  print_val("b", b);
  print_val("b - a", c);
  print_val("a - b", d);
  return 0;
}
```

When the above program is compiled with GCC 3.4.6 on a 64 bit Linux machine and run it produces the output:

```
a = 5
b = 7
b - a = 2
a - b = 18446744073709551614
```

In the above program, the subtraction of `a - b` is a programming error but that error results in the number `18446744073709551614` when using an unsigned type. Getting a number like `18446744073709551614` in program output or in the debugger does not exactly give a great hint as to what the problem might be. Was uninitialized memory used to produce this result? Is there some other memory usage problem that would cause the program to produce a ridiculous result such as this? It is problems like this that greatly contribute to the accidental complexity that is inherent in C/C++ programing (see Section 6.1).

However, when `unsigned long int` is replaced with `long int` in Listing 127 and the code is rebuilt and run one gets:

```
a = 5
b = 7
b - a = 2
a - b = -2
```

Now, getting output like `-2` when a positive number is expected is much easier to debug. The chance of getting `-2` as the result of a memory usage error is very unlikely. This would immediately be flagged as a subtraction error in the program and quickly tracked down and fixed. Therefore, from a program correctness and debugging perspective, signed integral types are far superior to unsigned types.

So if programs with unsigned integers are harder to debug when things go wrong, then what are the advantages of using an unsigned type? Well, some might argue that using an unsigned type for integral objects that that can only be non-negative in valid programs helps to make the code self documenting. This is partially true but one can achieve the same result by using a typedef to make the usage expectation clear (e.g. `size_type`).

So what then is left as the real advantage for using an unsigned integral type? The only real advantage of an unsigned integral type (e.g. `unsigned int`) over a signed integral type (e.g. `int`) is that unsigned integral type objects can represent twice the positive range as the equivalent signed integral type objects. For smaller integral types like `char` and and `short int`, having twice the range can be quite useful. However, on 32 bit and 64 bit modern computers, using an `unsigned [long] int` instead of a `[long] int` as the size for a container is quite worthless. On a 64 bit Linux machine with GCC 3.4.6, the sizes of several integral types pertinent to this discussion are shown in Listing 128.

**Listing 128** *: Sizes and ranges of some common integral types of GCC on a 64 bit Linux machine*

```
sizeof(int) = 4
std::numeric_limits<int>::min()= -2147483648
std::numeric_limits<int>::max()= 2147483647
std::log10(std::numeric_limits<int>::max())= 9.33193

sizeof(unsigned int) = 4
std::numeric_limits<unsigned int>::min()= 0
std::numeric_limits<unsigned int>::max()= 4294967295
std::log10(std::numeric_limits<unsigned int>::max())= 9.63296

sizeof(long int) = 8
std::numeric_limits<long int>::min()= -9223372036854775808
std::numeric_limits<long int>::max()= 9223372036854775807
std::log10(std::numeric_limits<long int>::max())= 18.9649

sizeof(unsigned long int) = 8
std::numeric_limits<unsigned long int>::min()= 0
std::numeric_limits<unsigned long int>::max()= 18446744073709551615
std::log10(std::numeric_limits<unsigned long int>::max())= 19.2659

sizeof(size_t) = 8
std::numeric_limits<size_t>::min()= 0
std::numeric_limits<size_t>::max()= 18446744073709551615
std::log10(std::numeric_limits<size_t>::max())= 19.2659
```

```
sizeof(ptrdiff_t) = 8
std::numeric_limits<ptrdiff_t>::min()= -9223372036854775808
std::numeric_limits<ptrdiff_t>::max()= 9223372036854775807
std::log10(std::numeric_limits<ptrdiff_t>::max())= 18.9649
```

On a 32 bit machine, `size_t` is a 4 bit `unsigned int` and `ptrdiff_t` is a 4 bit `int`. The standard C library typedef `size_t` is guaranteed to be the largest possible object size returned from `sizeof(...)` and is also used for functions like `malloc(...)`. The standard C library typedef `ptrdiff_t` is supposed to be guaranteed to hold the difference between the subtraction of any two pointers in the largest allocatable array. Right here lies the first problem with this approach as shown in the simple program in Listing 129.

**Listing 129** *: Simple program showing the fundamental incompatibility of `size_t` and `ptrdiff_t`.*

```
#include <iostream>
#include <string>
#include <limits>

template<typename T>
void print_val(const std::string &valName, const T val)
{ std::cout << valName << " = " << val << "\n";}

int main()
{
  const size_t maxSize = std::numeric_limits<size_t>::max();
  const size_t size = static_cast<size_t>(0.75 * maxSize);
  char *a = new char[size];
  ptrdiff_t a_diff = (a+size) - a;
  print_val("maxSize", maxSize);
  print_val("size", size);
  print_val("a+size - a", a_diff);
  delete a;
  return 0;
}
```

The program in Listing 129 allocates a `char` array 75% the size of the maximum allowed by `size_t`. In this program, `size` is 50% larger than the largest value that can be represented by the signed type `ptrdiff_t` (which is the type as `int` in this case). When this program is compiled with GCC 3.4.6 in 32 bit mode (i.e. with `-m32`) and run it produces the following output:

```
maxSize = 4294967295
size = 3221225471
a+size - a = -1073741825
```

The value of `ptr2 - ptr1 = -1073741825` where `ptr1 = a` and `ptr2 = a+size` is totally wrong. What this output suggests is that `ptr2 = a+size` is `1073741825` elements in front of `ptr1 = a` which is completely wrong when in actuality `ptr2 = a+size` is `size = 3221225471` elements after `ptr1 = a`. What this 32 bit output confirms is that it false to claim that `ptrdiff_t` can store the difference between

any two pointers in a single array of data. Perhaps that was true on the machines when C was first developed the early 1970's but it is not true today where machines with 4+ GB of memory are common.

Now consider practical usage of types like `std::vector` on modern machines. First, consider what it would mean to allocate the largest `std::vector` of even `char`s. A `char` is 1 byte so on a 32 bit machine, an `std::vector<char>` of max size would have `4294967295` bytes = 4.3 GB of memory. That would exhaust all of the memory of a 4 GB machine and more. Being limited to only half the range of `size_t` (which is the positive range representable by `ptrdiff_t`) would give an `std::vector<char>` that takes up 2.3 GB of memory. No real 32 bit program is ever going to allocate a single `std::vector` of `char`s that takes up more than half of the addressable memory of the entire machine! It is hard to imagine what useful task such a program would perform.

When one moves up to an `std::vector<int>` for a 32 bit (4 byte) `int` the maximum size of array that one can create is `4294967295 * 4 / 1e+9` = 17.2 GB. Being limited to the `unsigned int` type `ptrdiff_t` would limit one to an `std::vector<int>` of size 8.6 GB which is already twice the addressable memory of a 32 bit system.

Therefore, even on a 32 bit machine, limiting the maximum size of `std::vector` objects to have only `std::numeric_limits<ptrdiff_t)>::max()` = $2.3 \times 10^9$ elements is really not any kind of limit at all. For any reasonable program on any reasonable 32 bit machine one cannot even store that much memory.

On a 64 bit machine this of course becomes silly. By limiting the maximum number of elements in an `std::vector<char>` (not to mention arrays with larger data types) to be `std::numeric_limits<ptrdiff_t)>::max()` on 64 bit machine would mean that one would take up `18446744073709551615 / 2` = `9.2e+9` GB of memory to allocate a single array! We will likely never in human history ever see a machine with $2 \times 10^9$ GB of memory in a single address space.

In summary, limiting the maximum number of elements in an `std::vector` (and therefore Teuchos `Array`) to be half of `size_t` using the `long signed int` type `ptrdiff_t` for `size_type` is not any kind of limit at all in any realistic 32 bit program and especially not a 64 bit program.

Therefore, the Teuchos array memory management classes all use by default `ptrdiff_t` as `size_type` because of the inherent debugging and other advantages of using a signed integral type instead of an unsigned type and with no real advantages at all for using `size_t` over `ptrdiff_t`.

# D Raw performance data

## D.1 Raw RCP performance data

### Listing 130 *: Raw RCP timing data on GCC 4.1.2*

```
0. RCP_createDestroyOverhead_UnitTest ...

 Messuring the overhead of creating and destorying objects of different sizes
 using raw C++ pointers, shared_ptr, and using RCP.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.001 = 5e+06

   obj size   num loops  raw             shared_ptr      RCP             shared_ptr/raw  RCP/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          1    3465735    7.407462e-08    1.215497e-07    1.398462e-07    1.640909e+00    1.887910e+00
          4    2011797    7.450006e-08    1.239370e-07    1.413890e-07    1.663582e+00    1.897838e+00
         16     885379    8.031363e-08    1.284388e-07    1.467530e-07    1.599215e+00    1.827249e+00
         64     326124    1.130889e-07    1.589150e-07    1.792815e-07    1.405222e+00    1.585315e+00
        256     108380    2.369718e-07    2.786677e-07    2.359753e-07    1.175953e+00    9.957949e-01
       1024      33849    5.029395e-07    5.578008e-07    5.812875e-07    1.109081e+00    1.155780e+00
       4096      10153    1.552546e-06    1.608293e-06    1.630947e-06    1.035907e+00    1.050498e+00
      16384       2961    5.759541e-06    5.821344e-06    5.840932e-06    1.010731e+00    1.014132e+00
      65536        846    2.503073e-05    2.513239e-05    2.515721e-05    1.004061e+00    1.005053e+00

1. RCP_dereferenceOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240    7.765547e-10    1.037752e-09    6.958626e-10    8.960896e-01    6.705479e-01
         256    1083803    7.295887e-10    8.572714e-10    7.611003e-10    1.043191e+00    8.878173e-01
        1024     338498    7.117812e-10    8.238572e-10    7.125746e-10    1.001115e+00    8.649249e-01
        4096     101538    7.187575e-10    1.155846e-09    1.192136e-09    1.658607e+00    1.031397e+00
       16384      29614    8.350258e-10    1.155404e-09    1.190919e-09    1.426207e+00    1.030739e+00
       65536       8461    8.362559e-10    1.155293e-09    1.199785e-09    1.434711e+00    1.038512e+00

2. RCP_memberAccessOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240    7.794917e-10    1.037733e-09    6.954218e-10    8.921479e-01    6.701355e-01
         256    1083803    7.295743e-10    8.639896e-10    7.611075e-10    1.043221e+00    8.809221e-01
        1024     338498    7.115158e-10    8.325987e-10    7.242242e-10    1.017861e+00    8.698358e-01
        4096     101538    7.252928e-10    1.156058e-09    1.192244e-09    1.643811e+00    1.031302e+00
       16384      29614    8.369755e-10    1.154404e-09    1.190985e-09    1.422963e+00    1.031688e+00
       65536       8461    8.364092e-10    1.154440e-09    1.199527e-09    1.434139e+00    1.039056e+00

3. RCP_referenceCountManipulationOverhead_UnitTest ...

 Messuring the overhead of incrementing and deincrementing the reference count
 comparing RCP to raw pointer and boost::shared_ptr.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64      65224    1.554978e-09    4.145809e-09    6.009579e-09    3.864736e+00    1.449555e+00
         256      21676    7.138151e-10    4.221439e-09    5.832524e-09    8.170916e+00    1.381644e+00
        1024       6769    6.919181e-10    4.224365e-09    5.589158e-09    8.077773e+00    1.323076e+00
        4096       2030    6.863599e-10    4.226880e-09    6.094856e-09    8.879972e+00    1.441928e+00
       16384        592    6.854083e-10    4.224623e-09    6.234040e-09    9.095367e+00    1.475644e+00
       65536        169    6.848397e-10    4.228219e-09    6.216828e-09    9.077785e+00    1.470318e+00
```

**Listing 131** *: Raw RCP timing data on Intel ICC 10.1*

```
0. RCP_createDestroyOverhead_UnitTest ...

 Messuring the overhead of creating and destorying objects of different sizes
 using raw C++ pointers, shared_ptr, and using RCP.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.001 = 5e+06

   obj size   num loops  raw             shared_ptr      RCP             shared_ptr/raw  RCP/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          1    3465735    1.157942e-07    1.941906e-07    2.041379e-07    1.677032e+00    1.762938e+00
          4    2011797    1.194609e-07    1.984465e-07    2.031149e-07    1.661184e+00    1.700263e+00
         16     885379    1.200751e-07    2.013262e-07    2.105720e-07    1.676669e+00    1.753668e+00
         64     326124    1.390085e-07    2.170309e-07    2.277876e-07    1.561279e+00    1.638660e+00
        256     108380    3.299409e-07    4.036446e-07    4.223381e-07    1.223384e+00    1.280041e+00
       1024      33849    6.118349e-07    7.350291e-07    7.567432e-07    1.201352e+00    1.236842e+00
       4096      10153    1.724909e-06    1.833645e-06    1.851472e-06    1.063039e+00    1.073374e+00
      16384       2961    6.138467e-06    6.252955e-06    6.269504e-06    1.018651e+00    1.021347e+00
      65536        846    2.482151e-05    2.497754e-05    2.505437e-05    1.006286e+00    1.009381e+00

1. RCP_dereferenceOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240    6.909757e-10    3.551453e-09    1.027399e-09    1.486881e+00    2.892897e-01
         256    1083803    7.113406e-10    3.384829e-09    7.973226e-10    1.120873e+00    2.355577e-01
        1024     338498    6.914017e-10    2.841675e-09    7.658747e-10    1.107713e+00    2.695152e-01
        4096     101538    6.864300e-10    3.701316e-09    9.940787e-10    1.448187e+00    2.685744e-01
       16384      29614    7.319499e-10    3.367334e-09    9.934011e-10    1.357198e+00    2.950112e-01
       65536       8461    7.317167e-10    2.931704e-09    9.912099e-10    1.354636e+00    3.381003e-01

2. RCP_memberAccessOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240    6.899839e-10    6.952541e-10    8.252660e-10    1.196066e+00    1.186999e+00
         256    1083803    7.112650e-10    7.159684e-10    7.949727e-10    1.117688e+00    1.110346e+00
        1024     338498    6.911940e-10    6.924634e-10    7.741691e-10    1.120046e+00    1.117993e+00
        4096     101538    6.863435e-10    7.000054e-10    9.928501e-10    1.446579e+00    1.418346e+00
       16384      29614    7.326919e-10    8.507246e-10    9.953219e-10    1.358445e+00    1.169970e+00
       65536       8461    7.315725e-10    8.489935e-10    9.923713e-10    1.356491e+00    1.168880e+00

3. RCP_referenceCountManipulationOverhead_UnitTest ...

 Messuring the overhead of incrementing and deincrementing the reference count
 comparing RCP to raw pointer and boost::shared_ptr.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64      65224    1.032260e-09    5.619576e-09    8.805472e-09    8.530285e+00    1.566928e+00
         256      21676    7.246278e-10    5.881181e-09    8.692109e-09    1.199527e+01    1.477953e+00
        1024       6769    7.678041e-10    6.050677e-09    8.797574e-09    1.145810e+01    1.453982e+00
        4096       2030    7.621277e-10    5.988180e-09    8.991230e-09    1.179754e+01    1.501496e+00
       16384        592    8.004678e-10    6.102691e-09    8.966497e-09    1.120157e+01    1.469269e+00
       65536        169    7.999578e-10    6.108933e-09    8.973161e-09    1.121704e+01    1.468859e+00
```

**Listing 132** *: Raw RCP timing data on MSVC++ 2009*

```
0. RCP_createDestroyOverhead_UnitTest ...

 Messuring the overhead of creating and destorying objects of different sizes
 using raw C++ pointers, shared_ptr, and using RCP.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.001 = 5e+006

   obj size   num loops  raw             shared_ptr      RCP             shared_ptr/raw  RCP/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          1    3465735   2.628591e-007   3.641363e-007   4.117453e-007   1.385291e+000   1.566411e+000
          4    2011797   2.390897e-007   3.718069e-007   4.130635e-007   1.555094e+000   1.727651e+000
         16     885379   2.484812e-007   3.885342e-007   4.303242e-007   1.563636e+000   1.731818e+000
         64     326124   2.882339e-007   4.262182e-007   4.660804e-007   1.478723e+000   1.617021e+000
        256     108380   4.336593e-007   5.628345e-007   5.997416e-007   1.297872e+000   1.382979e+000
       1024      33849   9.749180e-007   1.093090e-006   1.122633e-006   1.121212e+000   1.151515e+000
       4096      10153   3.250271e-006   3.250271e-006   3.348764e-006   1.000000e+000   1.030303e+000
      16384       2961   1.182033e-005   1.350895e-005   1.215805e-005   1.142857e+000   1.028571e+000
      65536        846   4.609929e-005   4.609929e-005   4.609929e-005   1.000000e+000   1.000000e+000

1. RCP_dereferenceOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240   1.034882e-009   1.034882e-009   6.995039e-010   6.759259e-001   6.759259e-001
         256    1083803   1.052428e-009   1.052428e-009   7.136329e-010   6.780822e-001   6.780822e-001
        1024     338498   1.035711e-009   1.038596e-009   6.952820e-010   6.713092e-001   6.694444e-001
        4096     101538   1.021881e-009   1.043521e-009   1.012263e-009   9.905882e-001   9.700461e-001
       16384      29614   1.088221e-009   1.141807e-009   1.020207e-009   9.375000e-001   8.935018e-001
       65536       8461   1.082056e-009   1.141569e-009   1.011722e-009   9.350000e-001   8.862559e-001

2. RCP_memberAccessOverhead_UnitTest ...

 Messuring the overhead of dereferencing RCP, shared_ptr and a raw pointer.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64    3261240   1.039674e-009   1.044465e-009   1.015718e-009   9.769585e-001   9.724771e-001
         256    1083803   1.045220e-009   1.048824e-009   7.136329e-010   6.827586e-001   6.804124e-001
        1024     338498   1.032826e-009   1.038596e-009   6.923970e-010   6.703911e-001   6.666667e-001
        4096     101538   1.029094e-009   1.053139e-009   1.009859e-009   9.813084e-001   9.589041e-001
       16384      29614   1.077915e-009   1.135624e-009   1.007841e-009   9.349904e-001   8.874773e-001
       65536       8461   1.080252e-009   1.137962e-009   1.018936e-009   9.432387e-001   8.954041e-001

3. RCP_referenceCountManipulationOverhead_UnitTest ...

 Messuring the overhead of incrementing and deincrementing the reference count
 comparing RCP to raw pointer and boost::shared_ptr.

   array dim  num loops  raw             shared_ptr      RCP             RCP/raw         RCP/shared_ptr
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
          64      65224   7.186772e-010   5.270299e-009   1.413398e-008   1.966667e+001   2.681818e+000
         256      21676   3.604217e-010   4.144849e-009   1.261476e-008   3.500000e+001   3.043478e+000
        1024       6769   1.442698e-010   4.183825e-009   1.269575e-008   8.800000e+001   3.034483e+000
        4096       2030   2.405326e-010   4.089055e-009   1.274823e-008   5.300000e+001   3.117647e+000
       16384        592   3.092998e-010   4.227097e-009   1.278439e-008   4.133333e+001   3.024390e+000
       65536        169   2.708661e-010   4.153280e-009   1.291128e-008   4.766667e+001   3.108696e+000
```

## D.2 Raw Array performance data

### Listing 133 *: Raw Array timing data on GCC 4.1.2*

```
0. Array_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the Array braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560    4.244007e-10    4.244137e-10    4.244440e-10    1.000031e+00    1.000102e+00
         400     749245    3.631856e-10    3.629053e-10    3.629787e-10    9.992283e-01    9.994304e-01
        1600     230574    3.475457e-10    3.475999e-10    3.476270e-10    1.000156e+00    1.000234e+00
        6400      68470    5.450882e-10    5.452091e-10    5.367154e-10    1.000222e+00    9.846397e-01

1. Array_iteratorOverhead_UnitTest ...

 Measuring the overhead of the Array iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560    4.620638e-10    4.725251e-10    4.757363e-10    1.022640e+00    1.029590e+00
         400     749245    3.722247e-10    3.979673e-10    3.989416e-10    1.069159e+00    1.071776e+00
        1600     230574    3.498009e-10    3.796775e-10    3.797046e-10    1.085410e+00    1.085488e+00
        6400      68470    5.465578e-10    5.450813e-10    5.454967e-10    9.972986e-01    9.980585e-01

2. ArrayRCP_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    4.620552e-10    4.449722e-10    9.630283e-01
         400     749245    3.722748e-10    3.680972e-10    9.887783e-01
        1600     230574    3.498687e-10    3.486869e-10    9.966221e-01
        6400      68470    5.456381e-10    6.259333e-10    1.147158e+00

3. ArrayRCP_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    4.414750e-10    4.451065e-10    1.008226e+00
         400     749245    3.670995e-10    3.679571e-10    1.002336e+00
        1600     230574    3.485405e-10    3.488902e-10    1.001003e+00
        6400      68470    5.448531e-10    5.452068e-10    1.000649e+00

4. ArrayRCP_selfIteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP as a self iterataor relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    4.587616e-10    8.386087e-10    1.827984e+00
         400     749245    3.713705e-10    7.234583e-10    1.948077e+00
        1600     230574    3.497250e-10    6.945384e-10    1.985956e+00
```

```
        6400      68470    5.297461e-10    6.887003e-10    1.300057e+00

5. ArrayView_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the ArrayView braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    4.621072e-10    4.244570e-10    9.185250e-01
         400     749245    3.722814e-10    3.627618e-10    9.744291e-01
        1600     230574    3.499283e-10    3.474156e-10    9.928192e-01
        6400      68470    5.455994e-10    5.369824e-10    9.842065e-01

6. ArrayView_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayView iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    4.588396e-10    4.552254e-10    9.921232e-01
         400     749245    3.716074e-10    3.705230e-10    9.970818e-01
        1600     230574    3.495732e-10    3.493184e-10    9.992711e-01
        6400      68470    5.299196e-10    5.453255e-10    1.029072e+00
```

**Listing 134** *: Raw Array timing data on ICC 10.1*

```
0. Array_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the Array braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560    9.041672e-10    1.092201e-09    1.092210e-09    1.207964e+00    1.207973e+00
         400     749245    8.995689e-10    1.207742e-09    1.121359e-09    1.342579e+00    1.246551e+00
        1600     230574    8.816611e-10    1.154434e-09    1.172907e-09    1.309385e+00    1.330338e+00
        6400      68470    9.466212e-10    1.240822e-09    1.252366e-09    1.310790e+00    1.322986e+00

1. Array_iteratorOverhead_UnitTest ...

 Measuring the overhead of the Array iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560    6.093189e-10    6.334830e-10    5.921796e-10    1.039657e+00    9.718714e-01
         400     749245    6.862308e-10    6.941621e-10    6.795441e-10    1.011558e+00    9.902559e-01
        1600     230574    6.543805e-10    6.653910e-10    6.629027e-10    1.016826e+00    1.013023e+00
        6400      68470    7.261278e-10    7.312829e-10    7.259452e-10    1.007099e+00    9.997486e-01

2. ArrayRCP_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    7.565177e-10    1.091967e-09    1.443413e+00
         400     749245    7.061542e-10    1.116170e-09    1.580633e+00
        1600     230574    6.943432e-10    1.171088e-09    1.686613e+00
        6400      68470    6.937756e-10    1.305848e-09    1.882234e+00

3. ArrayRCP_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    3.658930e-10    3.765406e-10    1.029100e+00
         400     749245    3.789682e-10    3.671329e-10    9.687698e-01
        1600     230574    3.575045e-10    3.575994e-10    1.000265e+00
        6400      68470    5.485934e-10    5.484793e-10    9.997920e-01

4. ArrayRCP_selfIteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP as a self iterataor relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    3.729480e-10    1.878863e-09    5.037869e+00
         400     749245    3.919746e-10    1.754433e-09    4.475884e+00
        1600     230574    3.606841e-10    1.922398e-09    5.329866e+00
        6400      68470    5.474158e-10    2.262937e-09    4.133853e+00

5. ArrayView_braketOperatorOverhead_UnitTest ...
```

```
 Measuring the overhead of the ArrayView braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    7.635771e-10    1.092032e-09    1.430153e+00
         400     749245    7.155570e-10    1.121049e-09    1.566680e+00
        1600     230574    7.017405e-10    1.160129e-09    1.653217e+00
        6400      68470    7.770807e-10    1.261093e-09    1.622860e+00

6. ArrayView_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayView iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+03/0.0001 = 5e+07

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560    3.629461e-10    3.765493e-10    1.037480e+00
         400     749245    3.772431e-10    3.936763e-10    1.043561e+00
        1600     230574    3.589737e-10    3.622779e-10    1.009205e+00
        6400      68470    5.477992e-10    5.479590e-10    1.000292e+00
```

**Listing 135** *: Raw Array timing data on MSVC++ 2008*

```
0. Array_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the Array braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560   7.757111e-010   6.110350e-010   5.243634e-010   7.877095e-001   6.759777e-001
         400     749245   3.803829e-010   3.837196e-010   4.037398e-010   1.008772e+000   1.061404e+000
        1600     230574   3.523814e-010   3.605133e-010   3.550921e-010   1.023077e+000   1.007692e+000
        6400      68470   5.203009e-010   5.157368e-010   5.225829e-010   9.912281e-001   1.004386e+000

1. Array_iteratorOverhead_UnitTest ...

 Measuring the overhead of the Array iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         vector          Array           vector/raw      Array/raw
   ---------  ---------  --------------  --------------  --------------  --------------  --------------
         100    2307560   5.416977e-010   5.373641e-010   5.546985e-010   9.920000e-001   1.024000e+000
         400     749245   3.903930e-010   3.837196e-010   3.903930e-010   9.829060e-001   1.000000e+000
        1600     230574   3.550921e-010   3.794877e-010   3.659346e-010   1.068702e+000   1.030534e+000
        6400      68470   5.294289e-010   5.203009e-010   5.225829e-010   9.827586e-001   9.870690e-001

2. ArrayRCP_braketOperatorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560   5.330306e-010   8.753835e-010   1.642276e+000
         400     749245   4.938305e-010   8.642033e-010   1.750000e+000
        1600     230574   3.605133e-010   8.375836e-010   2.323308e+000
        6400      68470   5.317110e-010   8.169636e-010   1.536481e+000

3. ArrayRCP_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560   5.460313e-010   5.460313e-010   1.000000e+000
         400     749245   4.170865e-010   4.037398e-010   9.680000e-001
        1600     230574   3.578027e-010   3.578027e-010   1.000000e+000
        6400      68470   5.203009e-010   5.339930e-010   1.026316e+000

4. ArrayRCP_selfIteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayRCP as a self iterataor relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         ArrayRCP        ArrayRCP/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560   5.460313e-010   2.452807e-009   4.492063e+000
         400     749245   4.904938e-010   2.375725e-009   4.843537e+000
        1600     230574   3.578027e-010   2.355534e-009   6.583333e+000
        6400      68470   5.362750e-010   2.471429e-009   4.608511e+000

5. ArrayView_braketOperatorOverhead_UnitTest ...
```

```
 Measuring the overhead of the ArrayView braket operator relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560   5.330306e-010   5.286970e-010   9.918699e-001
         400     749245   3.803829e-010   4.004031e-010   1.052632e+000
        1600     230574   3.550921e-010   3.605133e-010   1.015267e+000
        6400      68470   5.111728e-010   5.180188e-010   1.013393e+000

6. ArrayView_iteratorOverhead_UnitTest ...

 Measuring the overhead of the ArrayView iterators relative to raw pointers.

 Number of loops = relCpuSpeed/relTestCost = 5e+003/0.0001 = 5e+007

   array dim  num loops  raw ptr         ArrayView       ArrayView/raw
   ---------  ---------  --------------  --------------  --------------
         100    2307560   5.373641e-010   5.460313e-010   1.016129e+000
         400     749245   3.970664e-010   3.970664e-010   1.000000e+000
        1600     230574   3.550921e-010   3.523814e-010   9.923664e-001
        6400      68470   5.294289e-010   5.408391e-010   1.021552e+000
```

DISTRIBUTION:

1 An Address
99 99$^{th}$ street NW
City, State
3 Some Address
and street
City, State
12 Another Address
On a street
City, State
U.S.A.

| | | |
|---|---|---|
| 1 | MS 1319 | Rolf Riesen, 1423 |
| 1 | MS 1110 | Another One, 01400 |
| 1 | M9999 | Someone, 01234 |
| 1 | MS 0899 | Technical Library, 9536 (electronic) |


Sandia National Laboratories